Capítulo 1

Introdução à Criptografia para Programadores: Evitando Maus Usos da Criptografia em Sistemas de Software

Alexandre Braga e Ricardo Dahab

Abstract

Studies have shown that vulnerabilities in cryptographic software are generally caused by implementation defects and mismanagement of cryptographic parameters. In addition, we see the recurring presence of several cryptographic bad practices in various software and mobile applications in particular. Possibly, these vulnerabilities were included unintentionally by inexperienced programmers without expert support. Along this vein, this short course addresses the programmatic use of cryptography by software developers with little or no experience in information security and cryptography. The material is introductory and aims to show software developers, through real examples and code snippets, the good and bad uses of cryptography and thus facilitate further improvements in future studies.

Resumo

Estudos têm revelado que vulnerabilidades em softwares criptográficos são causadas em geral por defeitos de implementação e pela má gestão de parâmetros criptográficos. Além disso, percebe-se a presença recorrente de diversas práticas ruins de criptografia em softwares diversos e aplicativos móveis em particular. Possivelmente, estas vulnerabilidades foram incluídas sem intenção por programadores inexperientes e sem apoio de especialistas. Desta forma, este minicurso aborda a utilização programática de criptografia por desenvolvedores de software com pouca ou nenhuma experiência em segurança da informação e criptografia. O material é introdutório e tem o objetivo de mostrar aos programadores de software, por meio de exemplos reais e trechos de código, os bons e maus usos da criptografia e, assim, facilitar o aprofundamento em estudos futuros.

 ©2015 SBC — Soc. Bras. de Computação

1.1. Introdução

Atualmente, a proliferação de smartphones e tablets e o advento da computação em nuvem estão mudando a forma como o software é desenvolvido e distribuído. Se por um lado há uma pulverização do esforço de construção de aplicativos, por outro surge, em escala sem precedentes, uma grande quantidade de aplicativos móveis disponíveis para aquisição em qualquer lugar e prontos para uso a qualquer momento.

Neste universo de aplicativos móveis sempre conectados, o uso de funções de segurança baseadas em técnicas criptográficas é cada vez maior. Observa-se que a utilização de criptografia tem aumentado muito, seja em termos do volume de dados criptografados (por exemplo, os smartphones mais modernos possuem sistemas de arquivos cifrados como uma opção padrão), seja em relação à quantidade de aplicativos com serviços criptográficos incluídos em seu funcionamento (por exemplo, uma loja de aplicativos pode vir a possuir centenas de aplicativos voltados a proteção criptográfica de dados: veja https://play.google.com/store/search?q=cryptography&c=apps).

Além dos casos de uso já tradicionais do software criptográfico autônomo, tais como encriptação/decriptação e assinatura/verificação, há diversas situações novas, intrinsecamente relacionadas à lógica da aplicação e do negócio, tornando cada vez mais diversificado o universo de ameaças ao software criptográfico moderno e exigindo cada vez mais do programador comum, geralmente não especializado em criptografia.

Este texto aborda a utilização programática de criptografia por desenvolvedores de software com pouca ou nenhuma experiência em segurança da informação e criptografia. O texto é introdutório e tem o objetivo de mostrar aos programadores de software, por meio de exemplos reais e trechos de código, os bons e maus usos da criptografia e, assim, facilitar o aprofundamento em estudos futuros.

Os objetivos do curso são os seguintes: *(i)* apresentar conceitos básicos de criptografia para programadores iniciantes em segurança da informação; *(ii)* mostrar como esses conceitos são utilizados em plataformas de desenvolvimento de software modernas; e *(iii)* ilustrar maus usos de criptografia comumente cometidos durante o desenvolvimento de software. Vale ainda ressaltar que o texto incentiva a utilização correta de implementações criptográficas prontas, a partir de uma API padronizada; por isto, a implementação de algoritmos criptográficos é tratada apenas superficialmente.

O tratamento dado ao tema é prático, a partir de programas de computador escritos na linguagem Java, com a API criptográfica padrão da plataforma Java [38] e a biblioteca criptográfica BouncyCastle [39]. Estas tecnologias foram escolhidas por serem amplamente utilizadas em uma variedade grande de plataformas de computação, desde sistemas corporativos baseados em serviços, sistemas web, até plataformas móveis modernas. Quando apropriado, exemplos reais, extraídos de incidentes causados por mau uso, são utilizados para ilustrar os conceitos apresentados. Os códigos fonte dos programas exemplo podem ser obtidos mediante solicitação aos autores.

O restante deste capítulo está organizado do seguinte modo. A Seção 1.2 explica conceitos básicos da criptografia; a Seção 1.3 mostra os usos adequados dos serviços criptográficos mais comuns; a Seção 1.4 explica os maus usos da criptografia que levam a vulnerabilidades em programas; a Seção 1.5 detalha a implementação em Java de um algoritmo criptográfico simétrico e a Seção 1.6 contém considerações finais.

1.2. Conceitos Básicos

As redes de comunicação abertas, como a Internet, geralmente não oferecem segurança intrínseca, fim-a-fim, para seus usuários. Não existe, por exemplo, sigilo intrínseco para a informação que viaja de um ponto a outro na grande rede. A criptografia é a única tecnologia capaz de garantir o sigilo e a autenticidade da informação em trânsito pelos meios eletrônicos. A criptografia pode ser usada de muitas maneiras, sendo muitas vezes a principal linha de defesa contra bisbilhotagem (*snooping*) e falsificação (*spoofing*).

A Criptografia (do grego *kryptos*, significando oculto) é a ciência que se dedica ao estudo e ao desenvolvimento das técnicas (matemáticas) utilizadas para tornar uma mensagem secreta. Historicamente, o verbo criptografar tem sido usado apenas nesse sentido. Entretanto, a criptografia moderna possui funções como assinaturas digitais, resumo (*hash*) criptográfico e outras, que não se limitam a prover sigilo da informação. De fato, como veremos a seguir, a palavra Criptografia denota hoje um conjunto de técnicas matemáticas das quais uma grande parte dos requisitos, mecanismos e serviços de segurança da informação não podem prescindir.

Esta seção aborda conceitos básicos e serviços criptográficos comumente encontrados em sistemas de software. São eles: objetivos e funções da criptografia; sistemas criptográficos e suas ameaças comuns; criptografia de chave secreta; criptografia de chave pública; encriptação para sigilo e privacidade; autenticação para identificação, irrefutabilidade de mensagens íntegras e autênticas; encriptadores de fluxo e de bloco; distribuição de chaves secretas; distribuição de chaves públicas; acordos de chaves; armazenamento seguro de chaves; gestão do ciclo de vida de chaves criptográficas (geração, distribuição, uso e revogação); chaves de sessão e sistemas criptográficos híbridos. Os livros [1][2][3][4] podem auxiliar no estudo do tema.

1.2.1. Objetivos e funções da criptografia

Historicamente associada ao sigilo, a criptografia moderna também oferece serviços para autenticação, integridade e irrefutabilidade. Os quatro serviços são os seguintes:

1. Confidencialidade (ou sigilo) é obtida com o uso da criptografia para manter a informação secreta, confidencial. Enviar e-mails encriptados e manter arquivos encriptados em cartões de memória são exemplos de confidencialidade.
2. Autenticação é obtida com o uso da criptografia para validar a identidade de uma entidade. Um exemplo de autenticação é o uso de assinaturas digitais para verificar a autoria de uma mensagem de texto ou de um documento eletrônico.
3. Integridade é obtida com o uso da criptografia para garantir que uma porção de dados não foi modificada desde a sua criação. Códigos de detecção de erros são exemplos de mecanismos para verificação de integridade de dados.
4. Irrefutabilidade é obtida pelo uso da criptografia como meio de garantir que o autor de uma mensagem autêntica não possa negar para um terceiro a sua autoria.

Na prática, estes serviços são usados juntos. Por exemplo, uma mensagem de correio eletrônico pode ser encriptada e assinada digitalmente. Deste modo, tanto a confidencialidade quanto a autenticação estarão garantidas. Visto que a assinatura digital é única para a mensagem, a integridade também é preservada.

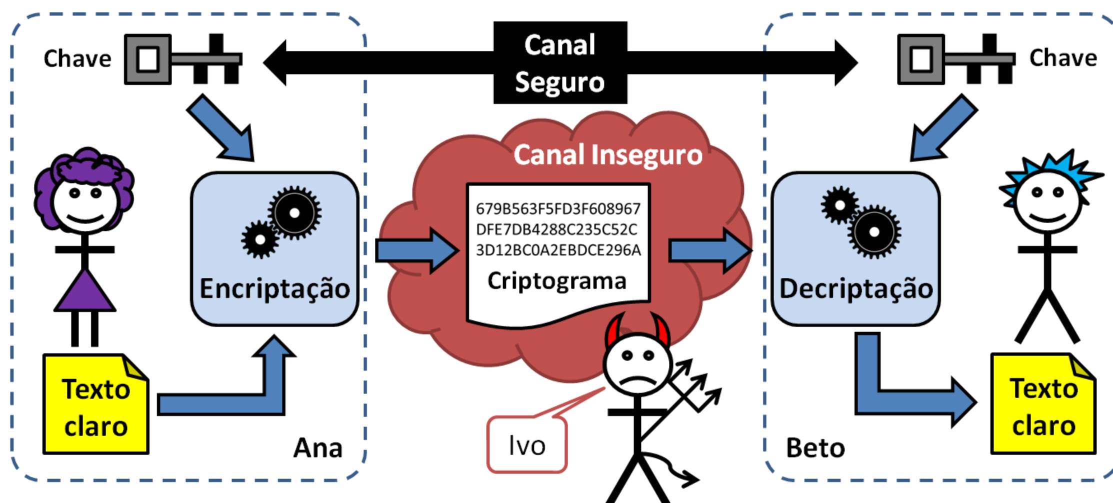

Figura 1: Sistema criptográfico.

1.2.2. Sistemas criptográficos

A Figura 1 mostra um sistema criptográfico e seus elementos fundamentais. Três personagens ilustram a figura: Ana, a remetente das mensagens; Beto, o destinatário das mensagens; e Ivo, o adversário com desejo de conhecer os segredos de Ana e Beto. As mensagens passam por um canal de comunicação inseguro e controlado por Ivo. O algoritmo criptográfico é usado para transformar o texto em claro (legível por qualquer um) em texto encriptado (o criptograma legível apenas por Ana e Beto) e vice-versa.

A chave criptográfica é o parâmetro de configuração do algoritmo que viabiliza a recuperação de um texto claro a partir do texto encriptado. Ana e Beto usam uma chave criptográfica conhecida apenas por eles e compartilhada (ou combinada) por um canal seguro diferenciado. Teoricamente, diz-se que a segurança do sistema criptográfico reside no segredo da chave e não no segredo do algoritmo criptográfico. Grosso modo, em sendo usado um algoritmo de boa reputação, a qualidade da implementação deste algoritmo e o tamanho da chave determinam a dificuldade em quebrar a encriptação da mensagem. A Figura 1 tem os seguintes passos:

1. Ana configura o algoritmo de encriptação com a chave compartilhada com Beto;
2. Ana passa o texto claro para o algoritmo e obtém o criptograma;
3. O criptograma é transmitido pelo canal inseguro e recebido por Beto;
4. Beto configura o algoritmo de decriptação com a chave compartilhada com Ana;
5. Beto decripta o criptograma recebido e obtém o texto claro original.

1.2.3. Ameaças comuns aos sistemas criptográficos

Encontrar vulnerabilidades em sistemas criptográficos, em vez de nas implementações destes sistemas, é uma tarefa complexa, pois normalmente os algoritmos criptográficos modernos são bem projetados, com segurança demonstrável, e submetidos ao escrutínio de pesquisadores por um bom período de tempo. Geralmente, o "teste do tempo" pode ser interpretado como evidência de robustez do algoritmo.

Em termos práticos, algoritmos criptográficos passam a ter valor a partir do momento em que são implementados, seja em software ou em hardware, para prover

confidencialidade, integridade, autenticidade e irrefutabilidade. Tradicionalmente, maior atenção tem sido dada à implementação confiável dos algoritmos criptográficos, uma vez que estas implementações podem expor problemas relacionados com o algoritmo subjacente, ou podem elas mesmas introduzir vulnerabilidades. Recentemente, tem crescido o interesse no uso correto dos algoritmos e suas implementações.

Uma implementação robusta de um sistema criptográfico é difícil de ser obtida, pois exige do desenvolvedor uma grande variedade de conhecimentos teóricos e práticos sobre criptografia, desenvolvimento de software seguro, arquitetura de computadores, compiladores, linguagens de programação, entre outras áreas da Ciência da Computação. Mesmo que o desenvolvedor possua esse tipo de conhecimento amplo e ao mesmo tempo profundo, defeitos de implementação ainda podem ocorrer.

Por causa destas dificuldades, em vez de tentar encontrar falhas nos algoritmos criptográficos, é mais fácil e prático para um adversário (Ivo) procurar por falhas não apenas nas implementações criptográficas, mas também, e às vezes principalmente, nos outros componentes dos sistemas criptográficos, como por exemplo, as camadas de software que encapsulam ou utilizam as implementações criptográficas.

Um ataque simples (porém, quase sempre impraticável) realizado por Ivo contra sistemas criptográficos é aquele conhecido como ataque de força bruta, no qual todas as possibilidades de chaves criptográficas são testadas na tentativa de decriptar corretamente o criptograma. Em geral, chaves longas são mais seguras, pois possuem um número maior de possibilidades. Vale observar que todos os outros ataques listados a seguir são facilitados se o primeiro ataque for bem sucedido e a chave secreta ou privada for comprometida (descoberta, adivinhada, ou deduzida). Ivo pode atacar um sistema criptográfico (por exemplo, um canal de comunicação protegido com criptografia) das seguintes maneiras:

1. Realizando um ataque de força bruta contra as chaves. Neste ataque, todas as chaves válidas possíveis são testadas na decriptação de um criptograma, para uma mensagem conhecida, até que a chave correta seja encontrada;
2. “Grampeando” o canal. Grampear um canal aberto é relativamente simples, pois basta ler a informação em trânsito. Por outro lado, para grampear um canal seguro é preciso não somente ler o criptograma em trânsito, mas também decriptá-lo. Para tal, seria necessário conhecer a chave de decriptação;
3. Reenviando mensagens antigas. Este ataque é possível se as mensagens não são unicamente identificadas (por exemplo, com carimbos temporais – *timestamps*) ou não possuem códigos de autenticação, ou ainda se as chaves não são trocadas periodicamente e com frequência adequada;
4. Personificando uma das partes comunicantes (Ana ou Beto). Ivo pode se fazer passar por Ana ou Beto pela substituição da chave de Ana/Beto pela sua própria, sem o conhecimento de Ana/Beto;
5. Assumindo o papel do intermediário (Man-in-the-Middle). Para este ataque, Ivo obtém as chaves de Ana e de Beto; personifica tanto Ana quanto Beto; intercepta os criptogramas de Ana/Beto para Beto/Ana; decripta estes criptogramas e os encripta novamente com sua própria chave de encriptação, antes de reenviá-los.

1.2.4. Tipos de sistemas criptográficos

Existem dois tipos de sistemas criptográficos, comumente conhecidos como criptografia de chave secreta (ou simétrica) e criptografia de chave pública (ou assimétrica). Na criptografia de chave secreta, uma única chave é usada para encriptar e decriptar a informação. Na criptografia de chave pública, duas chaves são necessárias. Uma chave é usada para encriptar; a outra chave, diferente, é usada para decriptar a informação. Estas duas chaves são matematicamente relacionadas e trabalham aos pares, de modo que o criptograma gerado com uma chave deve ser decriptado pela outra chave do par. Cada chave inverte o trabalho da outra e nenhuma pode ser usada sozinha em um sistema criptográfico. Nos sistemas de chave pública, uma das chaves do par é dita privada (a de decriptação), a outra é feita pública (a de encriptação).

1.2.5. Criptografia de chave secreta

Os sistemas criptográficos de chave secreta modernos possuem geralmente bom desempenho, mesmo em computadores considerados lentos. Com esta tecnologia, apenas uma chave é usada para encriptar e decriptar a informação. A Figura 2 ilustra os passos da encriptação com algoritmos simétricos de chave secreta, conforme a seguir:

1. Ana configura o algoritmo para o modo de encriptação com a chave secreta;
2. Ana alimenta o algoritmo com a mensagem original, o texto claro;
3. Ana encripta a mensagem e obtém o criptograma (mensagem encriptada).

Apenas a chave usada na encriptação pode decriptar corretamente a informação. Por isso, esta chave deve ser protegida e guardada em segredo; daí o nome de chave secreta. A Figura 2 também ilustra os passos da decriptação com chave secreta:

1. Beto configura o algoritmo para o modo de decriptação com a chave secreta;
2. Beto alimenta o algoritmo com a mensagem encriptada (criptograma);
3. Beto decripta a mensagem encriptada e obtém o texto claro original.

Teoricamente, este sistema criptográfico pode ser diretamente utilizado para encriptação bidirecional com a mesma chave. Porém, conforme tratado adiante neste texto, na prática, diversos detalhes de implementação dificultam a utilização segura da mesma chave para encriptação nas duas direções do canal de comunicação.

Na criptografia simétrica, a chave secreta deve ser conhecida por todos aqueles

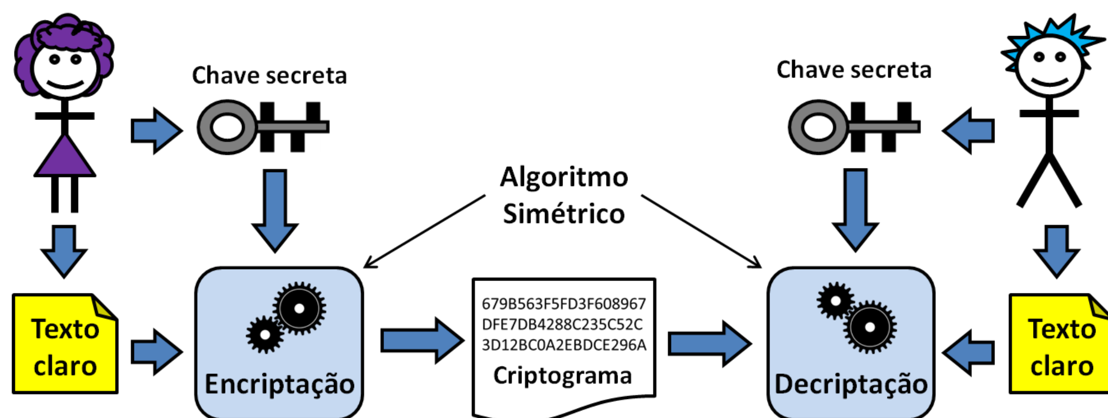

Figura 2: Sistema criptográfico simétrico.

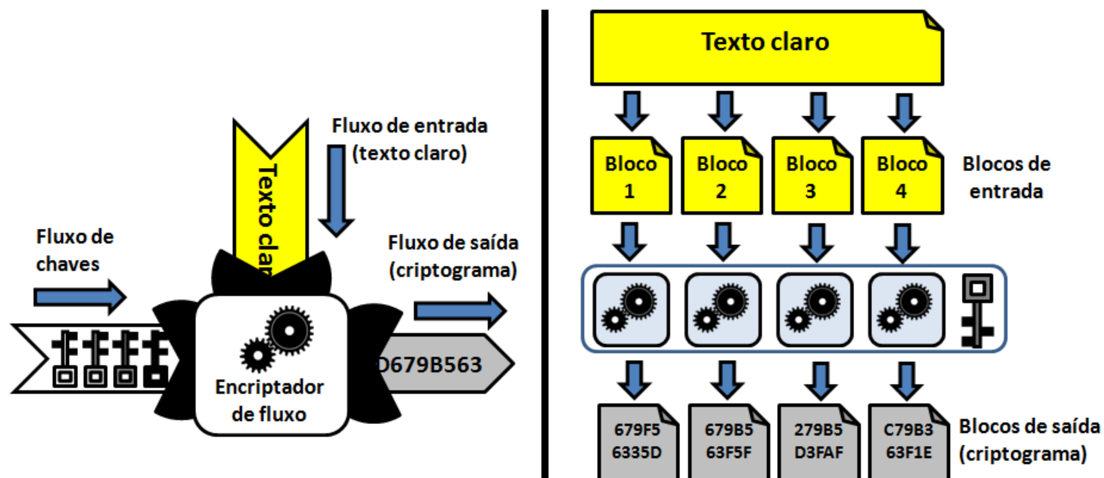

Figura 3: Encriptadores de fluxo (a esquerda) e de bloco (a direita).

que precisam decriptar a informação encriptada com ela. Mas como compartilhar a chave secreta sem que ela seja descoberta pelos adversários? Uma solução seria marcar um encontro secreto com todos que devem conhecer a chave. Porém, na Internet, isto não é fácil, afinal, usa-se a Internet quando encontros presenciais são difíceis. Sabe-se que a Internet é insegura por natureza (por isso, usar a criptografia), então não se pode simplesmente distribuir a chave secreta por e-mail ou mensagem de texto. A distribuição de chaves é um aspecto importante da criptografia de chave secreta. Apesar das dificuldades de distribuição de chaves, a criptografia de chave secreta é muito usada. Suas dificuldades aparentes podem ser contornadas pela combinação desta tecnologia com a criptografia de chave pública, originando sistemas criptográficos híbridos.

1.2.5.1. Encriptadores de fluxo e de bloco

Existem duas categorias de algoritmos simétricos (Figura 3): os encriptadores de fluxo e de bloco. Nos encriptadores de bloco, o texto claro é quebrado em blocos de bits de tamanho fixo. O encriptador trabalha sobre blocos e produz saídas em blocos também. O tamanho da chave criptográfica é geralmente um múltiplo do tamanho do bloco.

Os encriptadores de fluxo trabalham sobre sequências (de bits). A sequência ou fluxo de entrada é transformado continuamente na sequência ou fluxo de saída, bit a bit. É importante que a chave criptográfica seja uma sequência de bits pelo menos do mesmo tamanho do fluxo de entrada. Na prática, os bits da chave podem ser produzidos por um gerador de sequências de bits pseudoaleatórias, a partir de uma chave mestra de tamanho fixo.

1.2.6. Criptografia de chave pública

Devido à complexidade das operações matemáticas envolvidas, a criptografia de chave pública tradicional possui em geral um desempenho pior se comparada à criptografia de chave secreta, no mesmo hardware. A criptografia de chave pública utiliza duas chaves, que são relacionadas matematicamente e construídas para trabalharem juntas. Uma das chaves do par é dita a chave privada (pessoal) e é mantida em segredo, sendo conhecida

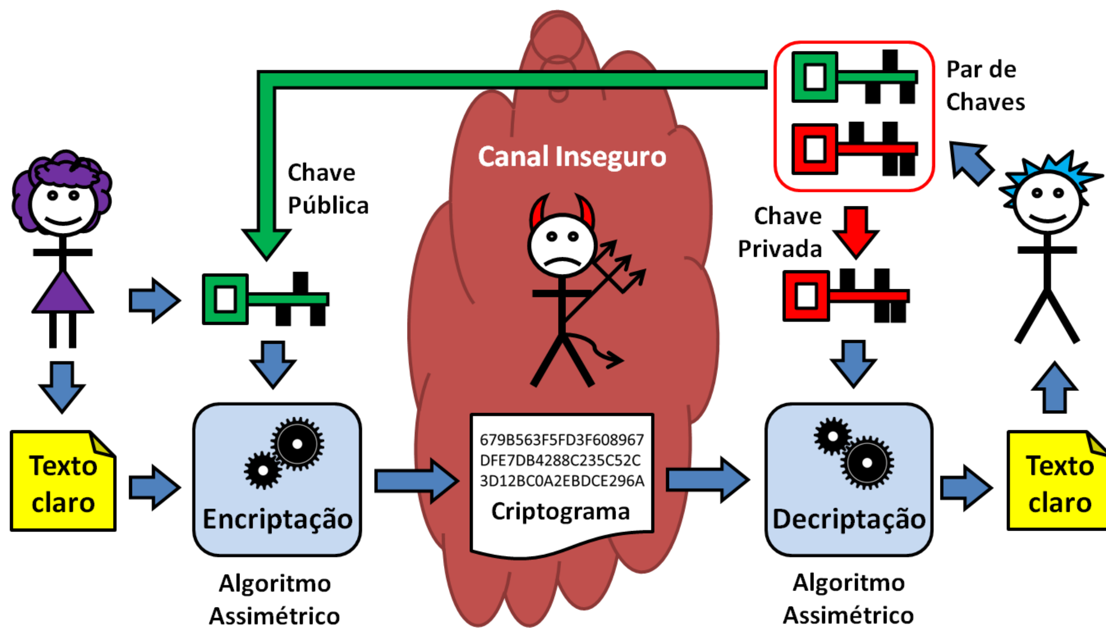

Figura 4: Sistema criptográfico assimétrico para sigilo.

apenas pelo dono do par de chaves. A outra chave do par é dita a chave pública por poder ser conhecida publicamente. A criptografia de chave pública pode ser usada para obter sigilo. Neste caso, a encriptação com a chave pública torna possível que qualquer um envie criptogramas para o dono da chave privada.

1.2.6.1. Encriptação para sigilo e privacidade

A Figura 4 ilustra um sistema criptográfico assimétrico para sigilo e seus elementos básicos. Mais uma vez, Ana, Beto e Ivo são os personagens. As mensagens de Ana para Beto são transmitidas por um canal inseguro, controlado por Ivo. Beto possui um par de chaves, uma chave pública e outra privada. Ana conhece a chave pública de Beto, mas somente o dono do par de chaves (Beto) conhece a chave privada (não há segredo compartilhado). A Figura 4 contém os passos a seguir:

1. Ana configura o algoritmo de encriptação com a chave pública de Beto;
2. Ana alimenta o algoritmo com a mensagem original (o texto claro);
3. O texto claro é encriptado e transmitido pelo canal inseguro para Beto;
4. Beto configura o algoritmo de decriptação com a sua própria chave privada;
5. O criptograma é decriptado e o texto claro original é obtido por Beto.

Analisando a Figura 4, observa-se que Ana envia uma mensagem privada para Beto. Para fazer isso, Ana encripta a mensagem usando a chave pública de Beto. Ana conhece a chave pública de Beto por que ela foi divulgada por Beto. Porém, o criptograma só pode ser decriptado pela chave privada de Beto; nem Ana pode fazê-lo.

Para obter comunicação segura bidirecional, basta acrescentar ao sistema criptográfico o mesmo processo no sentido oposto (de Beto para Ana), com outro par de chaves. Isto é, Beto usa a chave pública de Ana para enviar mensagens encriptadas para ela. Ana, ao receber a mensagem, usa sua própria chave privada pessoal para decriptar a mensagem enviada por Beto. A criptografia de chave pública é indispensável para a

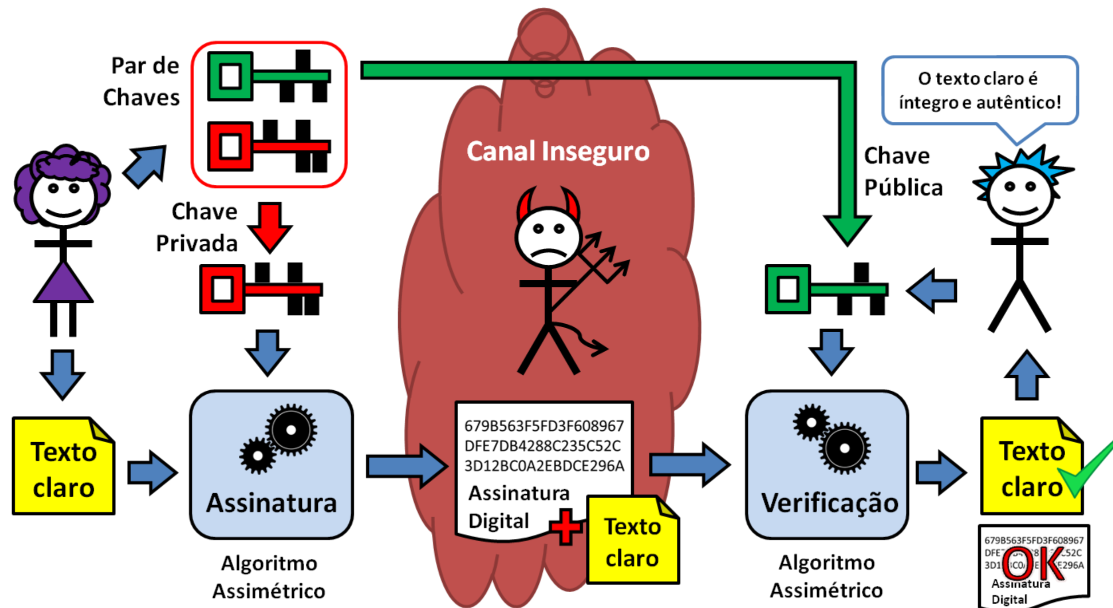

Figura 5: Sistema criptográfico assimétrico para autenticidade.

segurança da Internet, pois torna possível a comunicação privada em uma rede pública. A criptografia de chave pública é a base para outros dois serviços: a autenticação das partes e a verificação de integridade das mensagens.

1.2.6.2. Autenticação e irrefutabilidade

O uso da criptografia de chave pública para autenticação de mensagens é quase o oposto do uso para sigilo. A criptografia de chave pública para assinatura digital é usada para obter integridade, autenticidade e irrefutabilidade. Chama-se assinatura digital ao resultado de uma certa operação criptográfica com a chave privada sobre o texto claro. Neste caso, o dono da chave privada pode gerar mensagens assinadas, que podem ser verificadas por qualquer um que conheça a chave pública correspondente, portanto, sendo capaz de verificar a autenticidade da assinatura digital. Nem sempre a operação de assinatura é uma encriptação e a sua verificação é uma decriptação.

Visto que qualquer um de posse da chave pública pode “decriptar” a assinatura digital, ela não é mais secreta, mas possui outra propriedade: a irrefutabilidade. Isto é, quem quer que verifique a assinatura com a chave pública, sabe que ela foi produzida por uma chave privada exclusiva; logo, a mensagem não pode ter sido gerada por mais ninguém além do proprietário da chave privada.

Na Figura 5, Ana usa sua chave privada para assinar digitalmente uma mensagem para Beto. O texto claro e a assinatura digital são enviados por um canal inseguro e podem ser lidos por todos, por isso a mensagem não é secreta. Qualquer um que conheça a chave pública de Ana (todo mundo, inclusive Beto), pode verificar a assinatura digital. Ivo pode ler a mensagem, mas não pode falsificá-la, pois não conhece a chave privada de Ana.

O principal problema administrativo deste modelo de autenticação é justamente a confiança depositada na chave pública. Se a chave pública de alguém pode ser

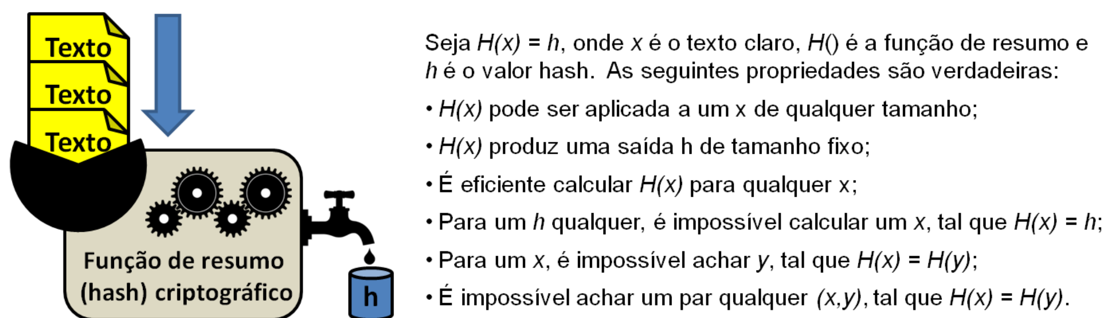

Figura 6: Funções de resumo (*hash*) criptográfico.

encontrada em qualquer lugar, então fica difícil saber se esta chave não foi corrompida ou substituída. O problema de garantir a autenticidade da chave pública é muito importante e, se não for solucionado satisfatoriamente, pode comprometer a confiança no sistema criptográfico. Uma maneira de validar chaves públicas é fazer com que elas sejam emitidas por Autoridades Certificadoras (AC) de uma Infraestrutura de Chaves Públicas (ICP), que torne possível a verificação da autenticidade de tais chaves.

1.2.6.3. Assinaturas digitais de tamanho fixo

A geração de assinaturas digitais com o mesmo tamanho do texto claro é ruim para a transmissão de dados. Além disso, a matemática envolvida na criptografia de chave pública é mais trabalhosa (complexa) e, por isso, não teria desempenho eficiente em processadores mais lentos. Por estas razões, normalmente, não é o texto claro inteiro que é assinado digitalmente, mas sim um resumo dele, que o identifique unicamente. Este identificador único é calculado por rotinas matemáticas chamadas de funções de resumo (ou *hash*) criptográfico, cujas propriedades são ilustradas na Figura 6.

Estas funções de *hash* geram uma sequência de bits, o valor do *hash*, que é único para o documento de entrada da função. O *hash* é muito menor que o documento original e geralmente tem um tamanho fixo de dezenas (algumas centenas) de bits. A função de *hash* é unidirecional porque não é reversível, isto é, não é possível recuperar o documento original a partir da sequência binária do *hash*. Além disso, idealmente, não existem dois documentos que geram o mesmo valor de *hash*.

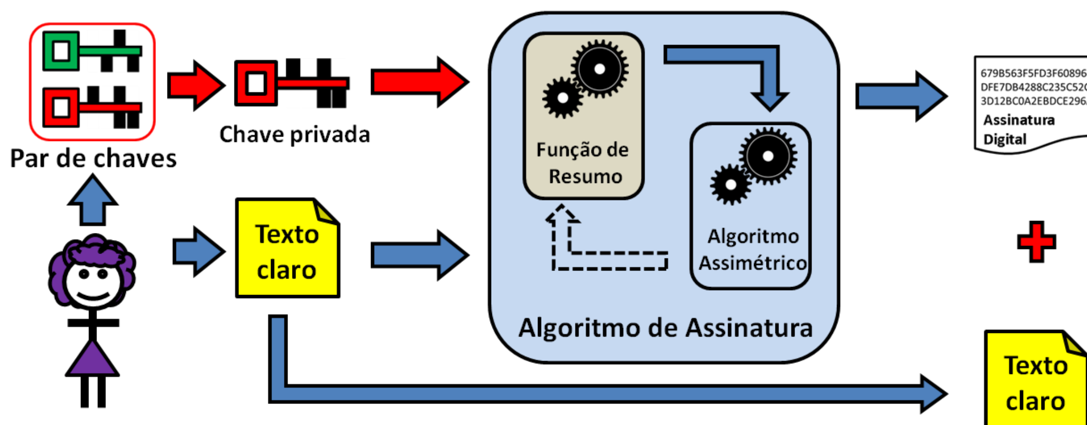

Figura 7: Assinaturas digitais de tamanho fixo.

Na Figura 7, Ana não assina o texto claro inteiro, pois ele pode ter um tamanho arbitrariamente grande. De fato, Ana usa um mecanismo de assinaturas digitais que calcula o *hash* do texto claro usando uma função de resumo criptográfico. O *hash* da mensagem é então assinado com a chave privada de Ana usando um mecanismo de assinatura digital. O modo de combinar a função de resumo e o algoritmo assimétrico é dependente de cada mecanismo de assinatura digital. Esta assinatura digital fixa é enviada junto com o texto claro original e está acessível para quem precisar verificar a autoria do texto claro.

A integridade do texto claro também está garantida, pois se ele tiver qualquer bit modificado, o *hash* calculado na verificação não corresponderá mais àquele da assinatura e a assinatura não será mais verificada com sucesso. Ana não pode mais negar (refutar) a autoria do texto claro, pois há uma assinatura digital feita com sua chave privada pessoal. Ninguém precisa da ajuda de Ana para verificar a autoria do documento, desde que a chave pública de Ana esteja amplamente disponível. Por isso, a assinatura digital é irrefutável.

1.2.7. Distribuição de chaves

A despeito da segurança oferecida pelas funções criptográficas para encriptação e assinatura digital, resta ainda um aspecto importante a ser considerado na análise de viabilidade de um sistema criptográfico: a distribuição de chaves. Distribuir as chaves na implantação de um sistema criptográfico significa oferecer a cada indivíduo participante do sistema todas as chaves necessárias para que ele consiga comunicar-se de modo seguro com qualquer outro participante. As tecnologias criptográficas simétricas e assimétricas possuem restrições diferentes quanto à distribuição de chaves.

Na criptografia simétrica, deve haver uma chave secreta compartilhada dentro de cada grupo de pessoas que deseja falar sem o conhecimento de outros. Por exemplo, para Ana e Beto trocarem segredos sem o conhecimento de Ivo, eles dois devem compartilhar uma chave secreta. A Figura 8 mostra a distribuição de chaves secretas entre pares para quatro indivíduos. Ana tem uma chave secreta para cada pessoa com que ela troca segredos sem que os outros saibam. Beto também quer ter seus segredos com cada pessoa em separado, por isso ele tem chaves secretas compartilhadas com cada participante. O mesmo vale para cada par de indivíduos.

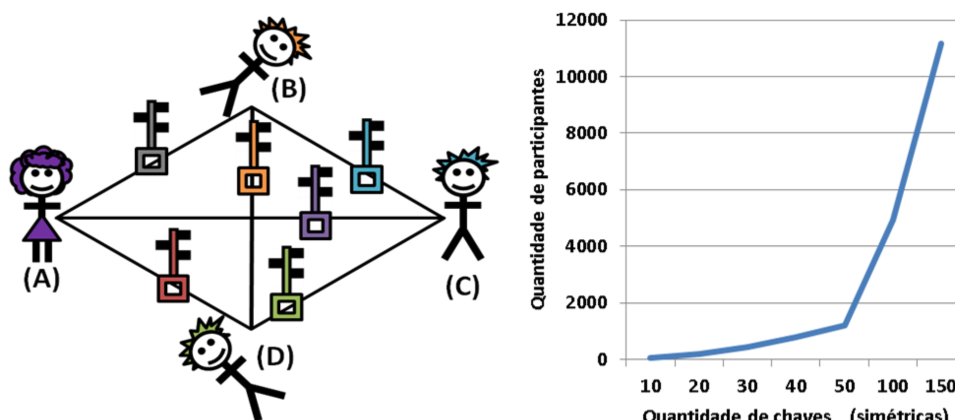

Figura 8: Distribuição de chaves simétricas e sua função de crescimento.

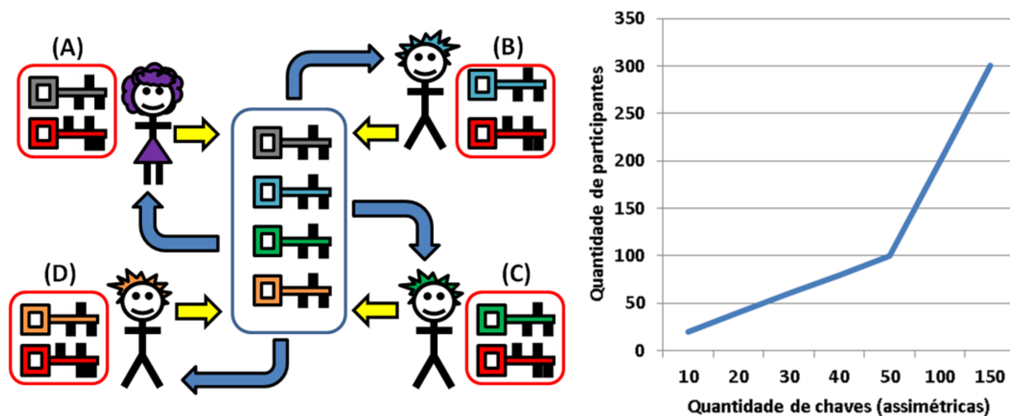

Figura 9: Distribuição de chaves assimétricas e sua função de crescimento.

Uma característica interessante dessa configuração é o sigilo mantido entre os pares. Cada par mantém seus segredos fora do alcance dos outros. O problema está na quantidade de chaves necessária para manter este nível alto de sigilo. Para um grupo de quatro pessoas, seis chaves são necessárias. Para cinco pessoas, dez chaves são necessárias, e assim por diante. Deste modo, este comportamento pode ser descrito por uma função quadrática. Isto é, a quantidade de chaves secretas cresce em função do quadrado da quantidade de pessoas do grupo.

O gráfico da Figura 8 mostra o crescimento da quantidade de chaves secretas em função do quadrado da quantidade dos participantes. Observa-se que para meros cem participantes, a quantidade de chaves secretas chega a alguns milhares, para cento e cinquenta participantes, tem-se mais de dez mil chaves e para duzentos participantes a quantidade de chaves quase chega a duzentos mil.

A criptografia assimétrica de chaves públicas simplifica muito o problema de distribuição de chaves. Cada participante só precisa ter o seu par de chaves, manter em segredo a chave privada e divulgar a chave pública. Pode haver um repositório global para todas as chaves públicas. A Figura 9 mostra a distribuição de chaves públicas para um grupo de quatro indivíduos. O gráfico da Figura 9 mostra o crescimento da quantidade de chaves (públicas e privadas) como uma função afim da quantidade de participantes. A quantidade de chaves é o dobro da quantidade de participantes.

1.2.8. Transporte de chaves e sistemas criptográficos híbridos

A criptografia de chave pública é recomendada para grupos grandes e ambientes dinâmicos e públicos. Por outro lado, a criptografia de chave secreta é recomendada para grupos pequenos e estáticos. A Tabela 1 compara as tecnologias criptográficas simétricas e assimétricas. Observa-se que as deficiências de uma tecnologia são complementadas pelas vantagens da outra. De fato, uma solução amplamente utilizada pelos protocolos de comunicação segura na Internet usa uma combinação das tecnologias simétrica e assimétrica, chamada de sistemas criptográficos híbridos.

Uma combinação interessante das tecnologias é aquela que oferece o desempenho superior da criptografia simétrica com a autenticação forte e a facilidade de distribuição de chaves da criptografia assimétrica. Tal combinação pode ser

implementada pelo uso de uma chave secreta encriptada por uma chave pública para transporte seguro. A criptografia assimétrica é usada como canal seguro para o compartilhamento da chave secreta. A Figura 10 mostra os passos da encriptação com a chave secreta em um sistema criptográfico híbrido:

1. Ana configura um algoritmo assimétrico com a chave pública de Beto;
2. Ana cria uma chave secreta e alimenta o algoritmo assimétrico com ela;
3. A chave secreta é encriptada pela chave pública de Beto;
4. Ana configura o algoritmo simétrico com a chave secreta;
5. Ana alimenta o algoritmo simétrico com um texto claro para Beto;
6. Ana envia o criptograma para Beto, junto com a chave secreta encriptada.

Durante uma comunicação segura, a sequência completa de passos para encriptação só é executada uma única vez, no início da comunicação, para compartilhar a chave secreta. A partir do segundo criptograma da conversa, basta que os passos 5 e 6 sejam executados. A tarefa de Beto é recuperar a chave secreta e manter a comunicação com Ana usando a chave secreta para proteger os dados.

De modo análogo, na decriptação, os passos para recuperar a chave secreta só são executados no início da comunicação. Observados os detalhes de implementação, depois que ambos (Ana e Beto) possuem a chave secreta, a comunicação pode ocorrer nos dois sentidos. Isto é, tanto de Ana para Beto, quanto de Beto para Ana.

Tabela 1: Comparação entre tecnologias criptográficas simétricas e assimétricas.

Criptografia Simétrica	Criptografia Assimétrica
Desempenho superior (mais rápida).	Desempenho inferior (mais lenta).
Não oferece autenticação forte, o segredo é compartilhado.	Autenticação forte com assinaturas digitais.
Não é irrefutável (a autoria pode ser negada).	Assinatura digital é irrefutável.
Distribuição de muitas chaves é trabalhosa.	Distribuição de muitas chaves é simplificada.

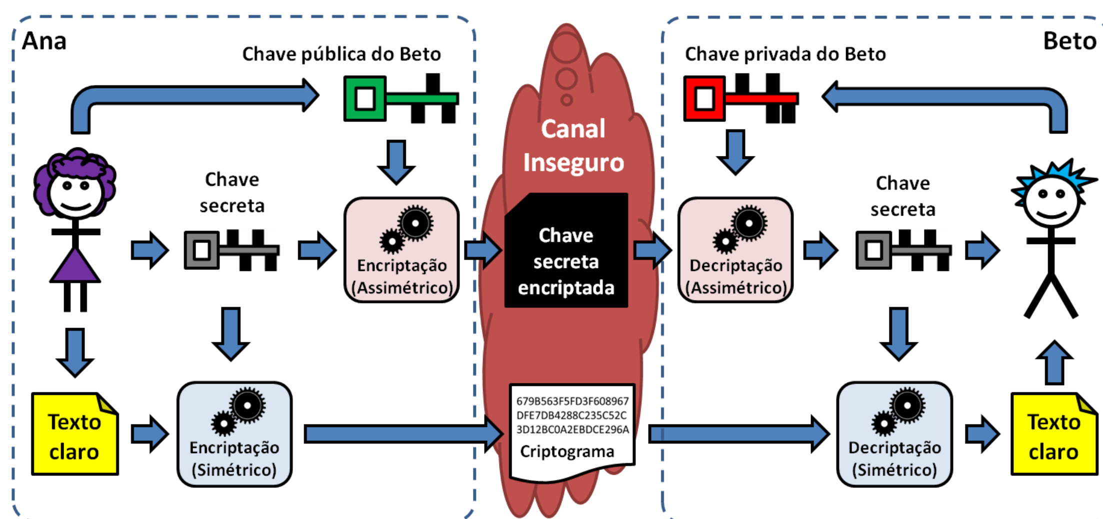

Figura 10: Sistema criptográfico híbrido e o transporte da chave de secreta.

1.2.9. Aspectos de gestão de chaves criptográficas

A gestão de chaves criptográficas é um tópico complexo e bastante suscetível a erros de construção, sendo muitas vezes a causa de vulnerabilidades graves em sistemas criptográficos. Esta seção aborda brevemente o armazenamento seguro de chaves em software, os métodos de acordo de chave e o ciclo de vida de chaves criptográficas.

1.2.9.1. Armazenamento seguro de chaves em software

O sigilo das chaves de encriptação e assinatura (privadas e secretas) é da maior importância para a segurança dos sistemas criptográficos. Nas implementações em software, sem auxílio de hardware de segurança, as chaves privadas e secretas devem ser guardadas de forma encriptada. A criptografia de proteção das chaves também necessita de chaves para encriptação, que por sua vez também precisam de proteção criptográfica, e assim por diante, levando a um problema aparentemente insolúvel.

A Criptografia Baseada em Senhas (*Password-Based Encryption* - PBE) resolve esta questão ao proporcionar os meios adequados para gerar uma chave criptográfica a partir de uma senha. A Figura 11 ilustra o funcionamento do método. O mecanismo de PBE é geralmente utilizado para proteger chaves logo que elas são criadas, para reduzir a exposição destas chaves. O mecanismo encapsula uma função de derivação de chaves (*Key Derivation Function* - KDF) e uma rotina de encriptação simétrica. Os passos de uso do PBE são os seguintes:

1. A PBE é configurada com uma senha (forte) e parâmetros de segurança;
2. A chave (privada ou secreta) a ser protegida é passada para a PBE;
3. Uma chave de proteção de chaves (CPC) é gerada pelo KDF a partir da senha;
4. A encriptação simétrica é configurada com a CPC;
5. A chave a ser protegida é encriptada e pode ser armazenada em um arquivo;
6. A CPC pode ser derivada novamente no futuro e por isto deve ser apagada.

A recuperação da chave encriptada é feita pela decriptação com a CPC derivada novamente a partir da senha. Vale ainda mencionar que as cópias em claro da chave a ser protegida devem ser apagadas do sistema, como uma boa prática de segurança. Mecanismos de verificação de integridade podem ser combinados ao método PBE.

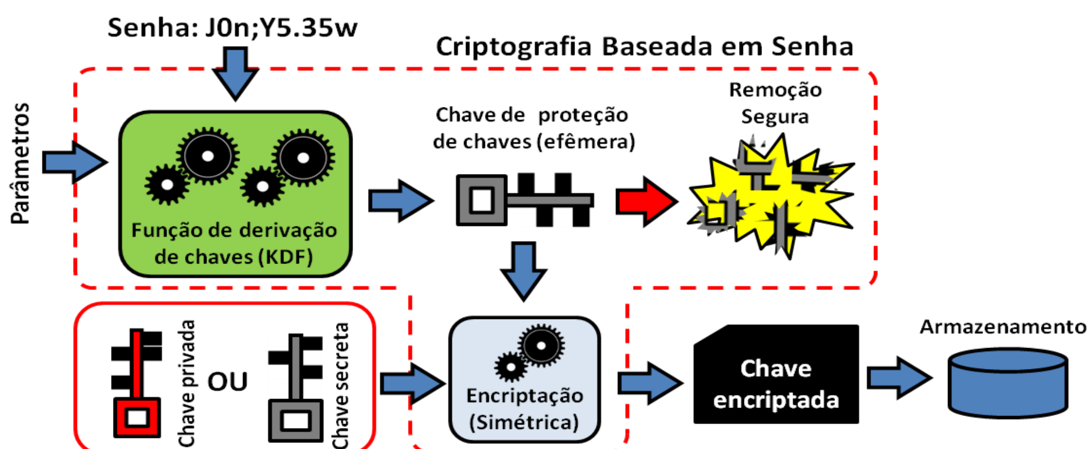

Figura 11: Armazenamento seguro de chaves com criptografia baseada em senhas.

1.2.9.1. Acordo de chaves

Há ocasiões em que entidades que nunca tiveram a oportunidade de compartilhar chaves criptográficas (por exemplo, nunca se encontram ou não se conhecem) precisam se comunicar em sigilo. Nestes casos, uma chave efêmera, usada apenas para algumas encriptações e decriptações decorrentes de uma conversa, pode ser gerada momentos antes do início da conversa. Os métodos de acordo de chaves são utilizados para combinar ou negociar uma chave secreta entre dois ou mais participantes usando um canal público. Uma característica interessante destes métodos é que o segredo compartilhado (a partir do qual a chave será derivada) é combinado pela troca de informações públicas por meio de um canal inseguro.

1.2.9.2. Ciclo de vida de chaves criptográficas

Uma chave criptográfica tem uma finalidade e está associada a uma entidade (pessoa ou sistema). A Figura 12 ilustra os estados e as fases da vida de uma chave criptográfica [28]. Uma chave possui seis estados, que são organizados em um ciclo de vida com quatro fases. Cada fase tem várias atividades específicas a serem realizadas. Em situações normais, uma chave passa a maior parte de sua vida útil no estado Ativado da fase Operacional. Uma chave desativada, que não está mais em operação, ainda pode ser utilizada para recuperar informação encriptada antiga.

O criptoperíodo é o período de tempo durante o qual as chaves para um determinado sistema permanecem em vigor e autorizadas para utilização. Uma chave sai de operação por dois motivos: o seu tempo de uso (criptoperíodo) terminou e ela será desativada, ou ela foi comprometida em algum incidente de segurança. A destruição de uma chave requer cuidados com a asseguração do apagamento ou remoção segura. Uma chave mal apagada ainda pode ser comprometida. Além disso, a destruição de uma chave comprometida não desfaz o comprometimento dos dados encriptados por ela, pois um segredo revelado nunca volta a ser um segredo de novo.

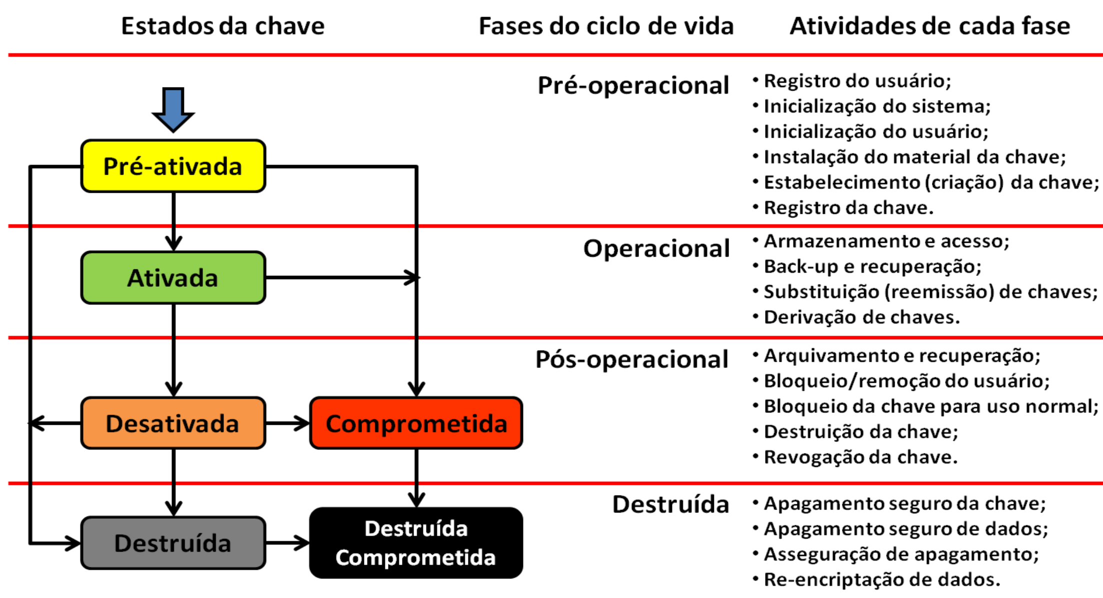

Figura 12: Estados e fases do ciclo de vida de uma chave criptográfica.

1.3. Como usar a criptografia: entendendo os bons usos

Esta seção aborda, por meio de (trechos de) programas em Java [38] e a biblioteca criptográfica BouncyCastle [39], os conceitos descritos na seção anterior. A bibliografia de apoio [10][11][12] explica aspectos específicos de programação da API criptográfica. Em particular, aqui são abordados os seguintes assuntos: encriptação/decriptação simétrica e assimétrica; modos de operação e suas propriedades; verificação de integridade e autenticação de mensagem; encriptação autenticada; criptografia baseada em senhas; transporte de chave simétrica com criptografia assimétrica; aleatoriedade e geração de números pseudoaleatórios; e distribuição e validação de certificados digitais. Ainda, os programas são inspirados nos padrões de projeto para criptografia [9].

1.3.1. O padrão de projeto de software criptográfico

A estrutura de um sistema criptográfico seque um padrão recorrente de arquitetura que foi primeiramente documento pelos autores em [9] e ficou conhecido como o padrão de projeto de software criptográfico. A Figura 13 mostra o diagrama de classes, em UML, de um sistema criptográfico simétrico para sigilo, em que Ana encripta e Beto decripta. O leitor deve ser capaz de abstrair a arquitetura de software da Figura 13 e visualizar estruturas semelhantes nos trechos de programas criptográficos mostrados a seguir.

Na figura, Ana e Beto usam a criptografia por meio de um aplicativo (App), em instanciações distintas, AnaApp e BetoApp, respectivamente. O App possui uma classe com estereótipo de controlador (CriptoCtrl) que é responsável por orquestrar os serviços criptográficos e a configuração de parâmetros de segurança juntamente com a lógica da aplicação. Por exemplo, no caso de Ana, o CriptoCtrl formata a mensagem *m* em um formato adequado para a criptografia, encripta o texto claro *tc* com a chave k_e e faz com que Beto receba o criptograma *c*. Beto, por sua vez, utiliza o seu orquestrador criptográfico para decriptar *c* com a chave k_d e converter o texto claro *tc* para a formatação requerida *m*. Ana e Beto utilizam instâncias diferentes da mesma classe Algoritmo (criptográfico).

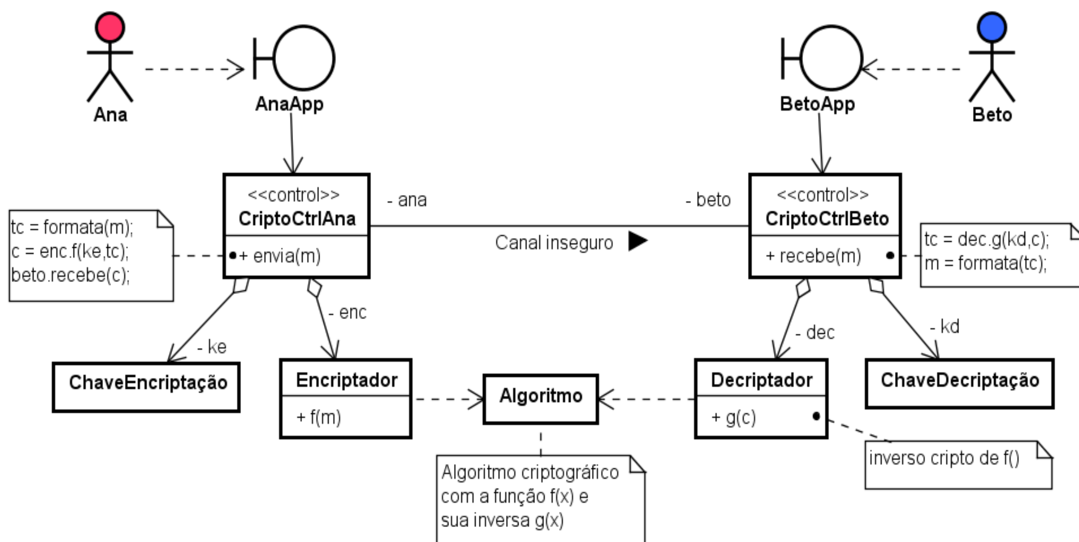

Figura 13: Sistema criptográfico simétrico para sigilo, em que Ana encripta e Beto decripta.

1.3.2. Encriptação e decriptação com chave secreta (simétrica)

O trecho de código mostrado no Programa 1 contém todos os elementos do sistema criptográfico simétrico usado por Ana e Beto. O Programa 1 não contém o código fonte completo, mas apenas o núcleo essencial para o sistema criptográfico. Trechos secundários, tais como importações de pacotes, declarações de variáveis auxiliares e apresentação de resultados, não são mostrados. Este estilo de apresentação será reproduzido no restante deste texto. O trecho de código do Programa 1 está dividido em três partes. A primeira parte, nas linhas de 01 a 06, contém as configurações comuns do sistema criptográfico. A segunda parte (linhas de 08 a 12) mostra o processo de encriptação pela Ana. A decriptação por Beto está nas linhas de 13 a 16.

As configurações comuns do sistema criptográfico são as seguintes. Na linha 02, o provedor criptográfico BouncyCastle (identificado adiante por “BC”) é adicionado dinamicamente. A configuração dinâmica é mais flexível e recomendada nas instalações em que não é possível, sem modificações do sistema operacional, alterar as configurações da Máquina Virtual Java (JVM), como é o caso dos dispositivos móveis Android. Nas linhas 03 e 04, um gerador de chaves é instanciado com o algoritmo criptográfico AES, do provedor “BC”, e inicializado para chaves de 256 bits. A chave criptográfica é gerada na linha 05. A linha 06 instancia o encriptador para o algoritmo AES como um encriptador de bloco no modo de operação CTR e *padding* PKCS#7 (os modos de operação e os mecanismos de *padding* são explicados adiante no texto).

O processo de encriptação começa na linha 09 com a configuração do encriptador para encriptação com a chave *k*. Na linha 11, os bytes do texto claro são encriptados pelo método `doFinal()`, produzindo o criptograma. O modo CTR requer um IV único, que não foi informado por Ana e por isto foi gerado automaticamente pelo encriptador e recuperado para uso da decriptação (linha 12). Na decriptação por Beto, o encriptador é configurado para decriptação com a chave compartilhada *k* e o mesmo IV usado na encriptação, linha 15. A decriptação (linha 16) devolve os bytes do texto claro e é direta pelo método `doFinal()`. O IV deve atender às propriedades do modo de operação CTR, sendo único e específico do criptograma, não devendo ser reutilizado com a mesma chave. Além do CTR, outros modos de operação comumente utilizados são o ECB, o CBC, o CFB e o OFB, detalhados adiante no texto.

Programa 1: Encriptação com criptografia simétrica com AES/CTR/PKCS7Padding.

```
01  // configurações do sistema criptográfico para Ana e Beto
02  Security.addProvider(new BouncyCastleProvider());
03  KeyGenerator g = KeyGenerator.getInstance("AES","BC");
04  g.init(256);
05  Key k = g.generateKey();
06  Cipher c = Cipher.getInstance("AES/CTR/PKCS7Padding", "BC");
07
08  // Encriptação pela Ana
09  c.init(Cipher.ENCRYPT_MODE, k);
10  byte[] textoclaroAna = "Testando o AES..".getBytes();
11  byte[] criptograma = c.doFinal(textoclaroAna);
12  byte[] iv = c.getIV();
13
14  // decriptação pelo Beto
15  c.init(Cipher.DECRYPT_MODE, k, new IvParameterSpec(iv));
16  byte[] textoclaroBeto = c.doFinal(criptograma);
```

1.3.3. Mecanismos de preenchimento de blocos incompletos

Os encriptadores de bloco trabalham sobre dados com tamanho igual a um múltiplo inteiro do tamanho do bloco. Já o texto claro tem tamanho livre. Para obter flexibilidade no texto claro, é necessário um mecanismo de preenchimento de blocos incompletos a fim de que o tamanho do texto claro seja múltiplo do tamanho do bloco. Chama-se de *padding* (preenchimento) o ato de completar a cadeia binária do texto claro para que ela tenha o tamanho múltiplo do tamanho do bloco do algoritmo de encriptação.

Os *paddings* PKCS#5, para blocos de 8 bytes, e PKCS#7, para blocos de 16 bytes, são calculados utilizando a mesma regra ilustrada a seguir para bloco de 16 bytes, em que M é o texto claro da mensagem, L é o tamanho de M em bytes, PM é a mensagem preenchida com o valor do *padding*, o caractere "|" é a concatenação e "resto" é o resto da divisão inteira de L por 16. Assim,

- Se o resto de L dividido por 16 é 15, PM = M | 0x01;
- Se o resto de L dividido por 16 é 14, PM = M | 0x0202;
- Se o resto de L dividido por 16 é 13, PM = M | 0x030303;
- Se o resto de L dividido por 16 é 12, PM = M | 0x04040404;
- Se o resto de L dividido por 16 é 11, PM = M | 0x0505050505;
- Se o resto de L dividido por 16 é 10, PM = M | 0x060606060606;
- Se o resto de L dividido por 16 é 09, PM = M | 0x07070707070707.

Assim por diante até a situação em que M tem um resto que é um bloco quase completo, exceto por um byte, e o resto de L dividido por 16 é 1. Nesta situação, a mensagem preenchida é PM = M | 0x0E0E0E0E0E0E0E0E0E0E0E0E0E0E. Finalmente, quando M tem tamanho múltiplo inteiro do bloco e o resto de L dividido por 16 é 0, temos PM = M | 0x0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F.

O trecho de código do Programa 2 mostra que o mecanismo de *padding* deve ser determinado no momento de instanciação do encriptador de bloco. Três mecanismos são ilustrados (PKCS#5, PKCS#7 e X9.23) além das opções sem *padding* e com *padding default*. Quando não há *padding*, somente blocos completos podem ser encriptados. Quando há *padding*, o criptograma é computado sobre o texto claro com *padding* e por isto, é maior que o texto claro original, podendo chegar a ter um bloco a mais.

Programa 2: Mecanismos de padding das cifras de bloco com PKCS5, PKCS7 e X9.23.

```
01  Cipher c1 = Cipher.getInstance("AES/ECB/NoPadding"   , "BC");
02  Cipher c2 = Cipher.getInstance("AES"                 , "BC");
03  Cipher c3 = Cipher.getInstance("AES/ECB/PKCS5Padding", "BC");
04  Cipher c4 = Cipher.getInstance("AES/ECB/PKCS7Padding", "BC");
05  Cipher c5 = Cipher.getInstance("AES/ECB/X9.23Padding", "BC");
```

```
01  // configurações do sistema criptográfico
02  Chave AES de 128 bits     : 073815E64B5F3B04F793B09813908994
03  Um bloco de texto claro   : Testando o AES..
04  Texto claro em hexadecimal: 54657374616E646F206F204145532E2E
05
06  // resultados da encriptação com padding
07  Nenhum : A0DF739B83DD6A72EFB4F5941B2A9915
08  Default:A0DF739B83DD6A72EFB4F5941B2A9915F0FFEC381B1CF8466000A69620F61A36
09  PKCS5  :A0DF739B83DD6A72EFB4F5941B2A9915F0FFEC381B1CF8466000A69620F61A36
10  PKCS7  :A0DF739B83DD6A72EFB4F5941B2A9915F0FFEC381B1CF8466000A69620F61A36
11  X9.23  :A0DF739B83DD6A72EFB4F5941B2A9915C378DA269D95F861666C9ABEA33AC9C5
```

1.3.4. Modos de operação dos encriptadores de bloco e suas propriedades

Os modos de operação dos encriptadores de bloco combinam de maneiras diferentes o IV e o texto claro (ou o criptograma) com o algoritmo criptográfico e a chave, para produzir encadeamentos de blocos encriptados característicos de cada um deles. Assim, o mesmo trio texto claro, chave e IV, quando usados com modos de operação diferentes, resultam em criptogramas diferentes. Os modos de operação mais usados são ECB, CBC, CFB, OFB e CTR [26]. Os modos ECB e CBC são encriptadores de bloco típicos. Os modos CFB, OFB e CTR se comportam como encriptadores de fluxo. Os modos de operação são diferentes em relação à recuperação de erros e à perda de sincronismo.

A Saída 1 mostra o resultado sobre o texto decriptado da modificação de um bit nos criptogramas gerados com os modos ECB, CBC, CFB, OFB e CTR, a partir da mesma chave e IV. A modificação é simplesmente o XOR do byte 1 com o valor 0x01. As linhas de 01 a 03 mostram a chave, o IV em hexadecimal e o texto claro, que tem tamanho de exatamente dois blocos do AES (32 bytes). No modo ECB, cada bloco é independente dos anteriores e não influencia seus sucessores. Por isto, na linha 07, o bloco do bit corrompido é perdido e o bloco seguinte é preservado.

O modo CBC faz o XOR do bloco processado pelo algoritmo com o bloco anterior ou com o IV, se for o primeiro bloco. Por isto, na linha 11, não apenas o bloco corrompido é completamente perdido, mas também o byte do segundo bloco que corresponde ao byte corrompido do primeiro bloco. Nos modos CFB, OFB e CTR, um fluxo de chaves é gerado pela encriptação encadeada (de maneiras específicas) da sequência de IVs. O XOR entre o fluxo de chaves e o texto claro produz o criptograma, e vice-versa. Na linha 15, com CFB, apenas o byte corrompido do primeiro bloco é perdido, já o bloco seguinte inteiro é perdido. Os modos OFB e CTR são análogos, em que apenas o byte corrompido do primeiro bloco é perdido e mais nada.

Em relação à perda de sincronismo, no modo ECB, a perda de um bloco do criptograma não afeta a decriptação dos blocos seguintes. Nos modos CBC e CFB, a perda de um bloco inviabiliza a decriptação apenas do bloco seguinte. Nos modos OFB e CTR, a perda e um bloco inviabiliza a decriptação de todos os blocos seguintes.

Saída 1: Comportamento dos modos de operação quando 1 bit do criptograma é modificado.

```
01   Chave      : 0123456789ABCDEF0123456789ABCDEF
02   iv         : ABCDEF1234567890ABCDEF1234567890
03   Texto claro: Modo de operacaoModo de operacao
04
05   Teste 1: AES/ECB/NoPadding
06   Criptograma: D005B98ACDC054C81666DB6B2EDF8D8BD004B98ACDC054C81666DB6B2EDF8D8B
07   Texto claro: ????????????????Modo de operacao
08
09   Teste 2: AES/CBC/NoPadding
10   Criptograma: 4A7F495EE367615DFC107C6B1A5589C70940086079FDB9D307D044C2E017D8D7
11   Texto claro: ????????????????Mndo de operacao
12
13   Teste 3: AES/CFB/NoPadding
14   Criptograma: F8241BBA339DC359EFA5ACF0DDB177583DBD525C351AA7388B95ADBF9E001926
15   Texto claro: Mndo de operacao????????????????
16
17   Teste 4: AES/OFB/NoPadding
18   Criptograma: F8241BBA339DC359EFA5ACF0DDB17758858195019B5F22A3D4FC3366EDC9095F
19   Texto claro: Mndo de operacaoModo de operacao
20
21   Teste 5: AES/CTR/NoPadding
22   Criptograma: F8241BBA339DC359EFA5ACF0DDB1775861AD5E88AE385B452C01C82A18A68E33
23   Texto claro: Mndo de operacaoModo de operacao
```

1.3.5. Verificação de integridade e autenticação de mensagem

O uso de encriptação somente não é capaz de garantir que o criptograma não foi corrompido, maliciosamente modificado, ou até mesmo completamente substituído por Ivo, quando em trânsito de Ana até Beto. Um valor de *hash* calculado sobre o criptograma poderia ser enviado por Ana junto com o criptograma e verificado por Beto. Deste modo, as corrupções acidentais ou maliciosas do criptograma poderiam ser detectadas por Beto. Mas Ivo ainda seria capaz de substituir o criptograma e o *hash*. Este problema é resolvido pelo uso de uma *tag* de autenticação no lugar do *hash*. Um código de autenticação de mensagem (MAC) é um mecanismo criptográfico que produz um selo (*tag*) de autenticidade baseado em uma chave compartilhada entre Ana e Beto e, por isto, relativa apenas a eles. A função de MAC recebe como entrada a chave simétrica e a mensagem e, geralmente, utiliza em seu algoritmo funções de encriptação ou de *hash*, produzindo a *tag*. A verificação da *tag* é feita pela comparação da *tag* recebida com uma nova *tag* calculada no recebimento da mensagem.

O trecho de código do Programa 3 mostra a utilização combinada de MAC e encriptação simétrica e está divido em três partes. As linhas de 01 a 06 contêm as configurações comuns do sistema criptográfico. As linhas de 08 a 11 mostram a encriptação e a geração da *tag* de autenticação pela Ana. A decriptação com a verificação por Beto da *tag* de autenticação está nas linhas de 13 a 16. A linha 02 mostra, em hexadecimal, o segredo compartilhado por Ana e Beto. A linha 03 produz a chave AES a partir do segredo. A linha 04 identifica o MAC como “HMACSHA256” [30], uma função de MAC baseada no *hash* (HMAC) da família SHA de 256 bits [29], e cria a chave correspondente. As linhas 05 e 06 instanciam o AES/ECB e o MAC.

A encriptação com *tag* de autenticação da mensagem ocorre do seguinte modo. Na linha 08, o MAC e o encriptador são inicializados com suas chaves, o criptograma é gerado na linha 10 e, na linha 11, a *tag* é calculada sobre o criptograma. A decriptação por Beto com verificação da *tag* ocorre assim: na linha 14, o MAC e o decriptador são inicializados com suas chaves, o criptograma é decriptado na linha 15 e, na linha 16, a *tag* recebida é comparada à *tag* calculada. A segurança do HMAC depende da função de *hash* e é dada pela metade de seu tamanho em bits. O HMACSHA256 tem *tag* de 256 bits e segurança de 128 bits; por isto as chaves do MAC e do AES também têm 128 bits.

Programa 3: MAC e encriptação com HMACSHA256 e AES/ECB/PKCS7Padding, 128 bits.

```
01  // configurações do sistema criptográfico para Ana e Beto
02  byte[] k = U.x2b("0123456789ABCDEF0123456789ABCDEF");
03  SecretKeySpec sks1 = new SecretKeySpec(k, "AES");
04  SecretKeySpec sks2 = new SecretKeySpec(k, "HMACSHA256");
05  Cipher c = Cipher.getInstance("AES/ECB/PKCS7Padding", "BC");
06  Mac m = Mac.getInstance("HMACSHA256", "BC");
07
08  // encriptação pela Ana com tag de autenticação da mensagem
09  m.init(sks2); c.init(Cipher.ENCRYPT_MODE, sks1);
10  byte[] criptograma = c.doFinal(textoclaroAna.getBytes());
11  byte[] tag = m.doFinal(criptograma);
12
13  // decriptação pelo Beto com verificação da tag
14  m.init(sks2); c.init(Cipher.DECRYPT_MODE, sks1);
15  byte[] textoclaroBeto = c.doFinal(criptograma);
16  boolean ok = MessageDigest.isEqual(m.doFinal(criptograma),tag);
```

1.3.6. Encriptação autenticada

A encriptação autenticada combina em uma única função criptográfica as funções de encriptação e de MAC. A união das duas funções em uma única rotina tem duas vantagens principais. A primeira é simplificar a programação do software criptográfico pelos programadores que, neste caso, utilizam uma API de mais alto nível. A segunda é evitar a combinação incorreta de encriptação e MAC (tratada na próxima seção). A encriptação autenticada não é apenas o encapsulamento das funções de encriptação e autenticação por uma API de mais alto nível. De fato, ela é uma função criptográfica nova que atende aos dois objetivos (encriptação e autenticação) simultaneamente. A função de encriptação autenticada é geralmente materializada por modos de operação específicos para encriptadores de bloco conhecidos, tais como os modos de operação GCM [27] do AES e CCM, aplicável a qualquer encriptador de bloco.

O trecho de código do Programa 4 mostra a encriptação autenticada com AES/GCM e está divido em três partes. As linhas de 01 a 04 contêm as configurações comuns do sistema criptográfico. As linhas de 06 a 09 mostram a encriptação autenticada com dados autenticados por Ana. A decriptação com a verificação por Beto está nas linhas de 11 a 15. A linha 02 já e conhecida e cria uma chave para o AES a partir de um segredo. A linha 03 constrói a estrutura para os parâmetros do modo GCM, o IV e o tamanho da *tag* de autenticação, com 128 bits (*tags* de 96 bits também são válidas). A linha 04 instancia o AES/GCM sem um *padding* explícito. O modo GCM é baseado no modo CTR e, por isto, segue as mesmas restrições do CTR para IVs únicos e não requer um *padding*, já que se comporta como um encriptador de fluxo.

O encriptador é inicializado com a chave e os parâmetros da encriptação autenticada na linha 07. A linha 08 mostra como dados adicionais que somente serão autenticados (AAD), mas não encriptados, são inseridos no encriptador pelo método `updateAAD()`. O criptograma é computado como de costume (método `doFinal()`) na linha 09. A *tag* de autenticação com 16 bytes é implicitamente anexada ao final do criptograma. Na linha 12, o encriptador e inicializado para decriptação autenticada e, na linha 13, ele recebe os dados adicionais autenticados, geralmente conhecidos por Ana e Beto. Na linha 14, a decriptação acontece com a verificação implícita e obrigatória da *tag*, que se for incorreta ou inválida, lança uma exceção especifica (linha 15) e impede a decriptação do criptograma, inviabilizando o uso de um texto claro não autenticado.

Programa 4: Encriptação autenticada com AES/GCM.

```
01  // configurações do sistema criptográfico para Ana e Beto
02  SecretKeySpec ks = new SecretKeySpec(k, "AES");
03  GCMParameterSpec gps = new GCMParameterSpec(128, iv);
04  Cipher c = Cipher.getInstance("AES/GCM/NoPadding", "BC");
05
06  // Encriptação pela Ana
07  c.init(Cipher.ENCRYPT_MODE, ks, gps);
08  c.updateAAD("AAD nao estah cifrado...".getBytes());
09  byte[] criptograma = c.doFinal(textoClaroAna.getBytes());
10
11  // decriptação pelo Beto
12  c.init(Cipher.DECRYPT_MODE, ks, gps);
13  c.updateAAD("AAD nao estah cifrado...".getBytes());
14  try {textoclaroBeto = c.doFinal(criptograma);}
15  catch (AEADBadTagException e) {ok = false;}
```

1.3.7. Criptografia baseada em senhas

Conforme já discutido anteriormente (Seção 1.2), a criptografia baseada em senhas (*Password-Based Encryption* - PBE) [21] oferece uma solução simples para um dos problemas mais comuns relacionados à gestão de chaves criptográficas em softwares criptográficos, a guarda segura de chaves criptográficas. A encriptação de chaves com chaves geradas a partir de senhas requer cuidados adicionais na escolha de senhas suficientemente fortes e outros parâmetros adequados. Em particular, uma senha ruim pode comprometer a segurança de todo o sistema criptográfico.

Senhas são normalmente menos seguras (menores, menos variadas, ou mais previsíveis) que chaves criptográficas. Por sito, a obtenção de uma chave boa a partir de uma senha qualquer (preferencialmente boa) requer decisões adequadas de segurança como, por exemplo, a escolha de uma boa função de derivação de chave (*Key Derivation Function* - KDF). A KDF utilizada pelo mecanismo PBE geralmente necessita, além da escolha de uma senha forte, da configuração de dois parâmetros de segurança: *(i)* um *nonce*, um número pseudoaleatório de uso único com a função de *salt* no KDF e *(ii)* um contador de iterações para a quantidade de vezes que a função criptográfica interna ao KDF será iterada sobre si mesma.

O trecho de código do Programa 5 mostra a utilização programática da API de PBE. Nas linhas 02 e 03, a senha e o *salt* são definidos. Na linha 04, a estrutura de parâmetros do PBE é instanciada e o contador é configurado para 2048. A boa prática de segurança estabelece um contador de pelo menos 1000 e um *salt* com pelo menos 32 bits [15]. As linhas 05 e 06 são bastante densas e estabelecem, de fato, a instância do PBE/KDF utilizado na derivação da chave criptográfica: um PBE sobre o *hash* SHA1 para uma chave do AES de 128 bits no modo CBC. Na linha 07, a chave é finalmente derivada de acordo com os parâmetros estabelecidos.

Uma vez tendo sido derivada, a chave pode ser utilizada normalmente conforme estabelecido no processo de derivação. No programa exemplo, a linha 08 cria um encriptador AES/CBC, que será utilizado por Ana para encriptação (linhas de 10 a 12) e por Beto para decriptação (linhas de 14 a 16) com a chave derivada. Neste caso, o segredo compartilhado entre Ana e Beto é a senha.

Programa 5: Criptografia baseada em senhas com PBEWithSHA1And128BitAES-CBC-BC.

```
01  // configurações do PBE comuns para Ana e Beto
02  char[] senha = "5enha!23".toCharArray();
03  byte[] salt = U.x2b("1234567890ABCDEF");
04  PBEKeySpec pbeks = new PBEKeySpec(senha, salt, 2048);
05  SecretKeyFactory skf = SecretKeyFactory.getInstance(
06                         "PBEWithSHA1And128BitAES-CBC-BC", "BC");
07  Key sk = skf.generateSecret(pbeks);
08  Cipher c = Cipher.getInstance("AES/CBC/PKCS7Padding", "BC");
09
10  // Encriptação pela Ana
11  c.init(Cipher.ENCRYPT_MODE, sk);
12  byte[] criptograma = c.doFinal(textoclaroAna.getBytes());
13
14  // decriptação pelo Beto
15  c.init(Cipher.DECRYPT_MODE, sk);
16  byte[] textoclaroBeto = c.doFinal(criptograma);
```

1.3.8. Encriptação e decriptação com chave assimétrica

Tradicionalmente, o algoritmo criptográfico assimétrico mais conhecido e utilizado para encriptação é o RSA, cujo nome é formado pelas letras iniciais dos sobrenomes dos autores Ron Rivest, Adi Shamir e Leonard Adleman. Para promover segurança e interoperabilidade, o uso e a implementação do RSA devem obedecer a padrões internacionais. O documento PKCS#1 v2.0 [22] especifica o *Optimal Asymmetric Encryption Padding* (OAEP) como um mecanismo de *padding*, que transforma o RSA em um mecanismo de encriptação assimétrica aleatorizado chamado RSA-OAEP.

As chaves criptográficas do RSA são muito grandes se comparadas às chaves de até 256 bits dos algoritmos simétricos, como o AES. Atualmente, tamanhos de chave RSA considerados seguros são 2048 bits ou 3072 bits, com um nível de segurança comparável ao dos algoritmos simétricos de 256 bits. A aritmética mais complexa e os tamanhos de chave elevados impõem ao RSA restrições de desempenho e de espaço.

O RSA-OAEP limita o tamanho do texto claro que pode ser encriptado em uma única chamada da função. Este limite de tamanho está relacionado ao tamanho do corpo finito (e da chave) usado na aritmética modular subjacente ao algoritmo e pode ser determinado, em bytes, pela fórmula $(ks\text{-}2*hs)/8-2$, onde ks é o tamanho da chave RSA em bits e hs é o tamanho do *hash* em bits usado pelo *padding* OAEP. Por exemplo, o RSA-OAEP com chave de 2048 bits e *hash* de 256 bits pode encriptar de uma vez um texto claro com até 158 bytes (1264 bits). O criptograma terá o tamanho da chave, que neste exemplo é de 256 bytes.

O trecho de código do Programa 6 mostra a configuração e uso do RSA-OAEP. A linha 02 instancia um gerador de par de chaves para o RSA. Na linha 03, o gerador é configurado para chaves de 2048 bits e o par de chaves é criado na linha 04. Nas linhas 07 e 08, a estrutura de parâmetros OAEP é criada com a função de mascaramento MGF1, o *hash* SHA-256 e a fonte de números primos padrão. As linhas 09 e 10 instanciam o RSA-OAEP com SHA256 e MGF1, que será utilizado por Ana para encriptação com a chave pública de Beto (linhas 13 e 14) e por Beto para decriptação com sua chave privada (linhas 17 e 18). Aqui, Ana já conhecia a chave pública de Beto.

Programa 6: Encriptação e decriptação assimétricas com RSA-OAEP.

```
01  // Beto cria um par de chaves
02  KeyPairGenerator g = KeyPairGenerator.getInstance("RSA", "BC");
03  g.initialize(2048);
04  KeyPair kp = g.generateKeyPair();
05
06  // configurações comuns para Ana e Beto
07  OAEPParameterSpec OAEPps = new OAEPParameterSpec("SHA-256",
08     "MGF1",MGF1ParameterSpec.SHA256, PSource.PSpecified.DEFAULT);
09  Cipher c = Cipher.getInstance(
10                "RSA/None/OAEPwithSHA256andMGF1Padding", "BC");
11
12  // Encriptação pela Ana com a chabe pública de Beto
13  c.init(Cipher.ENCRYPT_MODE, kp.getPublic(), OAEPps);
14  byte[] criptograma = c.doFinal(textoclaroAna.getBytes());
15
16  // Decriptação pelo Beto com sua chave privada
17  c.init(Cipher.DECRYPT_MODE, kp.getPrivate(), OAEPps);
18  byte[] textoclaroBeto = c.doFinal(criptograma);
```

1.3.9. Transporte de chave simétrica com encriptação assimétrica

As limitações de tamanho e de desempenho do RSA-OAEP podem ser minimizadas pela combinação deste algoritmo assimétrico com um algoritmo simétrico, como o AES, em um sistema criptográfico híbrido para transporte de chaves simétricas. O trecho de código do Programa 7 ilustra a encriptação de uma chave AES com RSA-OAEP.

O trecho de código é bastante semelhante aos anteriores, com as seguintes diferenças relevantes: as chaves RSA têm 3072 bits, o OAEP usa a função de *hash* SHA-512 e o AES está no modo CTR com *padding* PKCS#7 e chaves de 256 bits. Nesta configuração, o RSA encripta de uma única vez até 254 bytes, o que é mais do que suficiente para os 32 bytes da chave AES mais um bloco de *padding* PKCS#7.

Nas linhas 26 e 27, a chave secreta AES é encriptada por Ana com a chave RSA pública do Beto. Nas linhas 31 e 32, Beto decripta a chave secreta com sua chave RSA privada. A chave AES é restaurada (linha 35) e usada na decriptação (linhas 38 e 39).

Programa 7: Sistema criptográfico híbrido para transporte de chaves com RSA-OAEP e AES.

```
01  // Este é o par de chaves do Beto
02  KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA","BC");
03  kpg.initialize(3072);
04  KeyPair kp = kpg.generateKeyPair();
05
06  // configurações da criptografia assimétrica para Ana e Beto
07  OAEPParameterSpec OAEPps = new OAEPParameterSpec("SHA-512",
08     "MGF1",MGF1ParameterSpec.SHA512, PSource.PSpecified.DEFAULT);
09  Cipher x = Cipher.getInstance(
10                    "RSA/None/OAEPwithSHA512andMGF1Padding","BC");
11
12  // configurações da criptografia simétrica para Ana a e Beto
13  Cipher c = Cipher.getInstance("AES/CTR/PKCS7Padding", "BC");
14  IvParameterSpec ivps = new IvParameterSpec(iv);
15
16  // Chave secreta compartilhada entre Ana e Beto
17  KeyGenerator gAna = KeyGenerator.getInstance("AES", "BC");
18  gAna.init(256);
19  Key skAna = gAna.generateKey();
20
21  // Ana encripta alguma mensagem ...
22  c.init(Cipher.ENCRYPT_MODE, skAna, ivps);
23  byte[] criptograma = c.doFinal(textoclaroAna.getBytes());
24
25  // Encriptação por Ana da chave secreta com chave pública do Beto.
26  x.init(Cipher.ENCRYPT_MODE, kp.getPublic(), OAEPps);
27  byte[] chaveEncriptada = x.doFinal(skAna.getEncoded());
28
29  // Aqui começa a parte do Beto
30  // decriptação da chave secreta pelo Beto com a sua chave privada
31  x.init(Cipher.DECRYPT_MODE, kp.getPrivate(), OAEPps);
32  byte[] chaveBytes = x.doFinal(chaveEncriptada);
33
34  //recuperacao da chave secreta transportada
35  SecretKeySpec skBeto = new SecretKeySpec(chaveBytes, "AES");
36
37  //decriptando alguma coisa com a chave secreta
38  c.init(Cipher.DECRYPT_MODE, skBeto, ivps);
39  byte[] textoClaroBeto = c.doFinal(criptograma);
```

1.3.10. Assinaturas digitais e verificação de autenticidade

O trecho de código do Programa 8 mostra a utilização de dois sistemas criptográficos para assinaturas digitais. O primeiro é o RSA *Probabilistic Signature Scheme* (RSA-PSS), a padronização do RSA [22] para assinaturas digitais aleatorizadas. O segundo é o padrão norte-americano para assinaturas digitais sobre curvas elípticas, o ECDSA [25]. Os dois sistemas podem ser usados de modo alternativo, desde que a assinatura seja gerada e verificada pelo mesmo sistema.

O RSA-PSS foi configurado com SHA-256, MGF1 e chave de 3072 bits, produzindo assinaturas de 384 bytes. O ECDSA foi configurado com SHA-256 e a curva elíptica sobre corpo primo "`prime256v1`", produzindo uma assinatura digital de 70 bytes. A assinatura de tamanho reduzido e a maior velocidade de processamento, tanto na geração do par de chaves quanto na assinatura digital favorecem a tecnologia de curvas elípticas nas situações em que há restrições de espaço e de desempenho. O par de chaves do ECDSA também é consideravelmente menor em relação ao do RSA-PSS.

O par de chaves de Ana é gerado na linha 15. A geração de uma assinatura digital com a chave privada da Ana acontece nas linhas de 18 a 20. A verificação por Beto da autenticidade do documento com a assinatura de Ana, com a chave pública de Ana, ocorre nas linhas de 31 a 33. Beto já conhecia previamente a chave pública de Ana.

Programa 8: Dois algoritmos para assinaturas digitais: RSA-PSS e ECDSA.

```
01  // par de chaves de Ana e configurações do criptossistema
02  Signature sAna = null, vBeto = null;
03  KeyPairGenerator kpg = KeyPairGenerator.getInstance(alg,"BC");
04  switch (alg) {
05    case "RSA":
06      kpg.initialize(3072, new SecureRandom());
07      sAna = Signature.getInstance("SHA256withRSAandMGF1","BC");
08      break;
09   case "ECDSA":
10      ECGenParameterSpec ec= new ECGenParameterSpec("prime256v1");
11      kpg.initialize(ec, new SecureRandom());
12      sAna = Signature.getInstance("SHA256WithECDSA", "BC");
13      break;
14  }
15  KeyPair kpAna = kpg.generateKeyPair();
16
17  //Ana assina o documento
18  sAna.initSign(kpAna.getPrivate(), new SecureRandom());
19  assinadorAna.update(documento.getBytes);
20  byte[] assinatura = sAna.sign();
21
22  switch (alg) {
23    case "RSA":
24      vBeto = Signature.getInstance("SHA256withRSAandMGF1","BC");
25      break;
26    case "ECDSA":
27      vBeto = Signature.getInstance("SHA256WithECDSA", "BC");
28      break;
29  }
30  //Beto verifica a assinatura
31  vBeto.initVerify(kpAna.getPublic());
32  vBeto.update(documento.getBytes());
33  boolean ok = vBeto.verify(assinatura);
```

1.3.11. Aleatoriedade e geração de números pseudoaleatórios

O trecho de código do Programa 9 mostra a utilização de geradores de números pseudoaleatórios (*PseudoRandom Number Generators* - PRNGs). Em particular, o SHA1PRNG disponível no provedor criptográfico padrão do JDK é testado em relação a duas propriedades, a dispersão ou distribuição dos valores por um intervalo determinado e a previsibilidade da sequência pseudoaleatória gerada a partir de uma semente conhecida. Os PRNGs adequados para uso em sistemas criptográficos são ditos criptograficamente seguros. No Java, eles são objetos da classe SecureRandom.

Nas linhas de 02 a 04, as três instâncias do SHA1PRNG são criadas de modo independente. Nas linhas 06 a 08, as três instâncias produzem cada uma 100 valores inteiros no intervalo de 0 a 10 mil. O gráfico de dispersão da Figura 14(A) mostra os valores obtidos. Observa-se que os valores gerados pelas três instâncias estão distribuídos uniformemente pelo intervalo de 0 a 10 mil. Além disso, não houve repetição de valores nas sequências pseudoaleatórias. Este é o comportamento esperado de um PRNG criptograficamente útil. Testes mais rigorosos podem ser feitos [5].

Nas linhas 10 e 11, mais duas instâncias são criadas, mas desta vez elas são configuradas com a mesma semente pseudoaleatória (linha 12). Nas linhas 14 e 15, as duas instâncias produzem cada uma 100 valores inteiros no intervalo de 0 a 10 mil. O gráfico de dispersão da Figura 14(B) mostra os valores obtidos. Observa-se que as sequências pseudoaleatórias são idênticas. Para a mesma semente, o PRNG sempre produz a mesma sequência de valores. Este também é o comportamento esperado.

Programa 9: Geração de números pseudoaleatórios com SHA1PRNG.

```
01  // Teste de dispersão estatística
02  SecureRandom sr1 = SecureRandom.getInstance("SHA1PRNG", "SUN");
03  SecureRandom sr2 = SecureRandom.getInstance("SHA1PRNG", "SUN");
04  SecureRandom sr3 = SecureRandom.getInstance("SHA1PRNG", "SUN");
05  System.out.println("i , sr1 , sr2, sr3");
06  for (int i = 0; i < 100; i++) {
07    U.println(i+","+sr1.nextInt(10000)+","+sr2.nextInt(10000)+","
08                                        +sr3.nextInt(10000));}
09  //Teste de imprevisibilidade
10  SecureRandom sr4 = SecureRandom.getInstance("SHA1PRNG", "SUN");
11  SecureRandom sr5 = SecureRandom.getInstance("SHA1PRNG", "SUN");
12  byte[] s = sr4.generateSeed(32); sr4.setSeed(s); sr5.setSeed(s);
13  System.out.println("i , sr4 , sr5");
14  for (int i = 0; i < 100; i++) {
15    U.println(i+","+sr4.nextInt(10000)+","+sr5.nextInt(10000));}
```

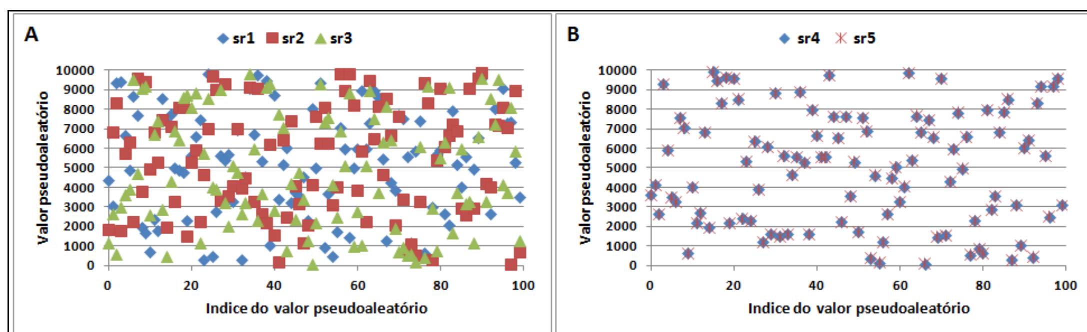

Figura 14: Testes do SHA1PRNG: (A) distribuição estatística e (B) imprevisibilidade.

1.3.12. Validação de certificados digitais

Certificação digital é um tema amplo e seu tratamento aprofundado tomaria todo o espaço deste texto, talvez merecendo até um capítulo exclusivo. Esta seção limita-se aos aspectos de validação de certificados digitais considerados boas práticas [7][8][28].

Um certificado digital de chave pública, ou simplesmente certificado, é um documento digital que dá como verdadeiro o vínculo entre uma chave pública autêntica e uma entidade cujo nome está no certificado. A veracidade do certificado é garantida por uma terceira parte confiável emissora do certificado, chamada de Autoridade Certificadora (CA). O certificado contém a assinatura digital da CA emissora e várias informações, tais como a chave pública, a identidade da entidade reconhecida pela CA, datas de início de uso e de validade (final de uso), etc. Por isto, o certificado é usado na verificação de que uma chave pública pertence a uma entidade e serve também como meio confiável para distribuição de chaves públicas.

A validação do certificado é realizada toda vez que a autenticidade da chave pública contida nele deve ser verificada. A assinatura da AC pode ser verificada por qualquer um com acesso à chave pública da AC, cujo certificado é amplamente disponível. Por exemplo, num caso bastante comum, uma AC pode emitir certificados de servidores web. Quando um software cliente HTTPS (browser) faz uma requisição para um servidor web protegido, o servidor responde com seu certificado digital. O software cliente valida o certificado do servidor verificando a assinatura da AC emissora sobre a chave pública do servidor e outros parâmetros do certificado. Se o cliente já não possuir a chave pública da AC, ele vai buscá-la (em um repositório de chaves públicas da AC). Se o certificado é válido, então o cliente sabe que o servidor é autêntico.

A validação do certificado (e da chave pública contida nele) abrange mais etapas do que a mera verificação da assinatura da AC. Além da verificação da assinatura, o software de verificação precisa verificar se o certificado não atingiu o final de seu período de validade, se não foi revogado e se o nome constante no certificado é o

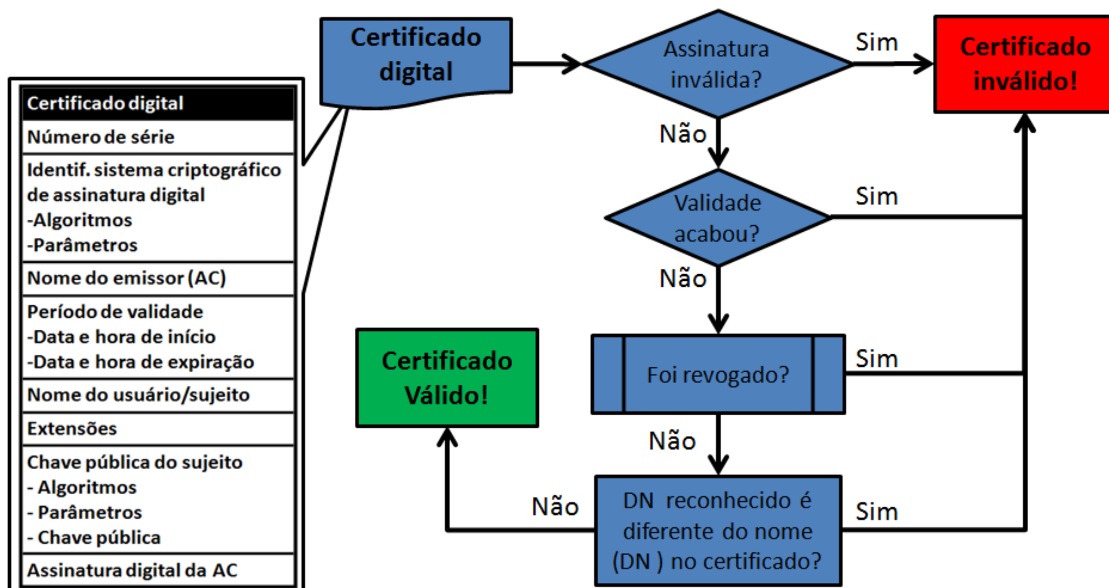

Figura 15: Fluxograma de validação de um certificado digital.

mesmo da parte que alega ser a dona da chave pública. O nome também deve ser obtido de um terceiro confiável, por exemplo, de um serviço de DNS, no caso de nomes de domínio. A validação do certificado é ilustrada no fluxograma da Figura 15.

A verificação da assinatura da AC em um certificado digital exige a chave pública da AC. Para ser confiável, a chave pública da AC deve estar contida em um certificado assinado por outra AC ou autoassinado, se for uma CA raiz. As verificações sucessivas de uma sequência de assinaturas constroem uma hierarquia de certificados. Os certificados na base da hierarquia são assinados pelas ACs de mais baixo nível, cujos certificados são assinados pelas ACs intermediárias, que têm seus certificados assinados pelas ACs de alto nível, cujos certificados são assinados pela AC raiz, que tem seus certificados autoassinados. A Figura 16 ilustra esta cadeia de certificação.

Uma AC revoga um certificado nas seguintes situações: quando ocorre erro na emissão do certificado (nome grafado errado), ou o certificado foi emitido para uso de um serviço e o portador não tem mais acesso a ele (demissão de um funcionário), ou a chave privada do portador foi comprometida, ou ainda a chave privada da AC foi comprometida (uma situação extrema). Uma Lista de Certificados Revogados (LCR) é o documento assinado digitalmente pela AC que lista o número de série de todos os certificados, ainda não expirados, que perderam a utilidade por algum dos motivos acima. Um software criptográfico pode consultar um serviço de LCR para receber atualizações periódicas, em intervalos regulares definidos por procedimentos, ou ainda consultar em tempo real se um certificado foi revogado ou não. Porém, a instabilidade de comunicação pode causar indisponibilidade do serviço de validação em tempo real.

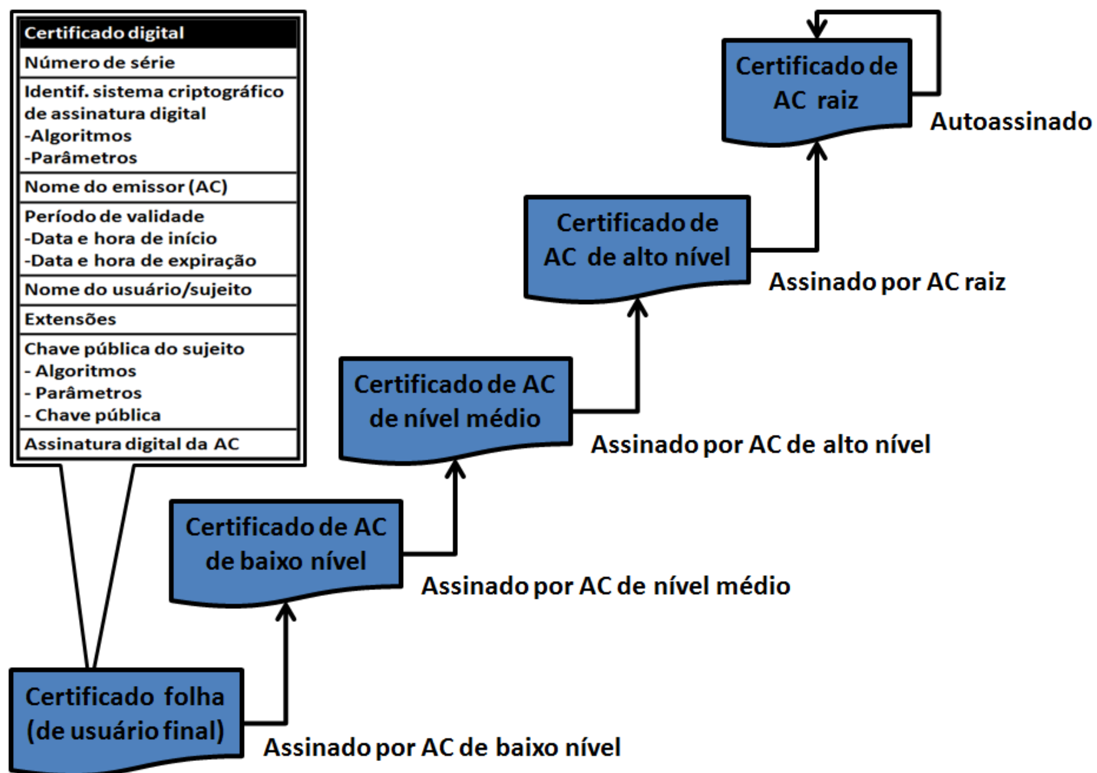

Figura 16: Cadeia hierárquica de certificação digital.

1.4. Como não usar a criptografia: reconhecendo os maus usos

Esta seção é organizada em torno dos maus usos de programação de criptografia que levam a vulnerabilidades em softwares criptográficos [13][17][18] e que têm sido estudados recentemente [14][15][20]. Os maus usos são exemplificados por casos reais e/ou ilustrados programaticamente por meio de programas em Java. Os maus usos comuns de criptografia tratados no curso são os seguintes: criptografia determinística simétrica e assimétrica, IVs fixos ou reutilizados, sementes fixas ou reutilizadas para PRNGs, troca indevida de modos de operação, combinação de integridade e encriptação, reutilização de chaves em encriptadores de fluxo, maleabilidade dos encriptadores de fluxo, *padding oracles* e validação incorreta de certificados digitais. A utilização de chaves fracas (pequenas) e de algoritmos obsoletos ou quebrados não é tratada aqui como um mau uso programático, mas sim uma configuração mal feita.

1.4.1. Criptografia determinística simétrica

A criptografia determinística simétrica é caracterizada pelo uso indiscriminado do modo de operação ECB juntamente com um encriptador simétrico, com ou sem *padding*. A utilização do modo ECB é considerada um mau uso por que, quando é utilizada inadvertidamente, pode levar a revelação indevida de informação pela identificação de padrões do texto claro no criptograma. Neste caso, a facilidade com que a criptografia determinística pode ser utilizada também favorece o seu mau uso.

Em Java, este mau uso pode ser identificado já na instanciação do algoritmo criptográfico, por exemplo, pelo método *getInstance(a)*, da classe *Cipher*, em que *a* é o nome do algoritmo. Há três opções para resolver o nome do algoritmo *a* com o modo ECB: *(i)* apenas o nome do algoritmo, por exemplo, *"AES"*; *(ii)* nome do algoritmo e modo sem *padding*, por exemplo, *"AES/ECB/NoPadding"*; e *(iii)* nome do algoritmo e modo com *padding*, por exemplo, *"AES/ECB/PKCS7Padding"*. Vale observar que a primeira opção leva naturalmente ao erro, uma vez que, na falta de uma escolha explícita feita pelo programador, o modo ECB é a opção padrão implícita. A Saída 2 mostra o resultado da encriptação do texto claro *"Deterministica.."* nas três opções de configuração do modo ECB, com o algoritmo AES e a mesma chave criptográfica. Na linha 04 pode ser observado que a encriptação do mesmo texto claro com os mesmos parâmetros de segurança produz o mesmo criptograma com *padding* que as outras opções de encriptação determinística (mostradas nas linhas 08 sem *padding*, e 11 com *padding* explícito). O algoritmo AES foi utilizado apenas como ilustração; o mesmo resultado seria obtido com outros encriptadores de bloco nas mesmas configurações de chave e modo de operação. Vale lembrar que o modo ECB não necessita de IV.

Saída 2: Criptografia determinística com AES no modo ECB.

```
01   Texto claro: Deterministica..
02   Chave   AES: C64539B4A9E992A077170C413FC02EB2
03
04   Encriptado com: AES
05   Criptograma: EAB196E34C4ED4B8C4261CACC243AC5E507D286A609456C69DA7CE68C844B89E
06
07   Encriptado com: AES/ECB/NoPadding
08   Criptograma: EAB196E34C4ED4B8C4261CACC243AC5E
09
10   Encriptado com: AES/ECB/PKCS7Padding
11   Criptograma: EAB196E34C4ED4B8C4261CACC243AC5E507D286A609456C69DA7CE68C844B89E
```

1.4.2. Criptografia determinística assimétrica

A criptografia determinística assimétrica é caracterizada pelo uso indiscriminado do algoritmo RSA na sua forma canônica, isto é, sem aleatorização. Analogamente ao modo ECB dos encriptadores simétricos de bloco, a utilização do algoritmo RSA sem *padding* aleatorizado é considerada um mau uso porque, quando é utilizada inadvertidamente, pode levar a revelação indevida de informação pela identificação de padrões do texto claro no criptograma. Mais uma vez, a facilidade com que a criptografia determinística pode ser utilizada também favorece o seu mau uso [17].

Em Java, este mau uso pode ser identificado já na instanciação do algoritmo criptográfico, por exemplo, pelo método *getInstance(a)*, da classe *Cipher*, em que *a* é o nome do algoritmo. Há três opções para resolver o nome do algoritmo *a* para o RSA canônico: *(i)* apenas o nome do algoritmo, por exemplo, *"RSA"*; *(ii)* nome do algoritmo e modo ECB sem *padding*, por exemplo, *"RSA/ECB/NoPadding"*; e *(iii)* nome do algoritmo, sem modo e sem *padding*, por exemplo, *"RSA/None/NoPadding"*. Vale observar que a primeira opção leva naturalmente ao erro, uma vez que, na falta de uma escolha explícita, o RSA canônico é a opção padrão implícita.

A Saída 3 mostra o resultado da encriptação do texto claro *"Cripto deterministica"* nas três opções de configuração do RSA canônico e a mesma chave criptográfica de 512 bits (uma chave pequena útil somente em exemplos). Este tamanho de chave foi escolhido apenas para facilitar a apresentação do resultado e não tem relação com este mau uso. Na linha 04 pode ser observado que a encriptação do mesmo texto claro, com a mesma chave, pelo algoritmo identificado por "RSA" produz o mesmo criptograma que as outras opções de encriptação determinística com RSA e são mostrados nas linhas 08, *"RSA/ECB/NoPadding"*, e 12, *"RSA/None/NoPadding"*. A Saída 3 também mostra as duas implementações de RSA aleatorizados disponíveis no provedor BouncyCastle: a versão mais antiga do padrão PKCS#1 identificada por "RSA/None/PKCS1Padding" e o PKCS#1 versão 2 conhecido como RSA-OAEP e identificado por "RSA/None/OAEPWithSHA1AndMGF1Padding". Ambos mostram criptogramas diferentes não apenas entre si, mas também dos anteriores.

Saída 3: Criptografia determinística assimétrica com RSA.

```
01   Texto claro: Cripto deterministica
02
03   Encriptado com: RSA
04   Criptograma: 6C77C92E46A4D7A110F3713D8B635BE7677E140CC607DEF342F10A8CD0AC00E7
05                E865439CF2501D6CDADAF8884CE4B61C6BA91E13225E5AB375A9DE2662059C3F
06
07   Encriptado com: RSA/ECB/NoPadding
08   Criptograma: 6C77C92E46A4D7A110F3713D8B635BE7677E140CC607DEF342F10A8CD0AC00E7
09                E865439CF2501D6CDADAF8884CE4B61C6BA91E13225E5AB375A9DE2662059C3F
10
11   Encriptado com: RSA/None/NoPadding
12   Criptograma: 6C77C92E46A4D7A110F3713D8B635BE7677E140CC607DEF342F10A8CD0AC00E7
13                E865439CF2501D6CDADAF8884CE4B61C6BA91E13225E5AB375A9DE2662059C3F
14
15   Encriptado com: RSA/None/PKCS1Padding
16   Criptograma: 6528EFA1281661E80D83728E5A1FEFDB55E45525DF9027C36784AF0BB375DB84
17                01767A45DD070B0A85FB31687785314A86C5DA89267259961EFF1D681891F08B
18
19   Encriptado com: RSA/None/OAEPWithSHA1AndMGF1Padding
20   Criptograma: 225EDA0DE625CF0F06400985FF1933F6F4457690F4513F4A0F4547B49A68F9C7
21                7CD600412BE0A4147D97DC030D3CAEF30FC47B8069A53540F818D36D61944ED6
```

1.4.3. IVs fixos ou reutilizados

O mau uso chamado de reutilização ou fixação dos Vetores de Inicialização (IVs) é caracterizado pela realização de mais de uma encriptação com o mesmo IV, para a mesma chave criptográfica. A repetição de IVs com a mesma chave é considerada um mau uso por que pode levar à identificação de padrões do texto claro no criptograma. Este mau uso é facilmente identificado em programas de computador quando um valor fixo para o IV está embutido no código fonte. Já ocorrências cuja identificação é mais desafiadora são aquelas em que o IV é obtido de modo indireto, por exemplo, de um arquivo ou banco de dados, ao ainda computado a partir de algoritmos determinísticos, como por exemplo, um PRNG de semente fixa.

A Saída 4 mostra o resultado da encriptação do texto claro *"Teste de IV fixoTeste de IV fixo"* de 32 bytes (dois blocos do AES), com a mesma chave e IV. Para cada modo de operação (CBC, OFB, CFB e CTR) foram feitas duas encriptações sem *padding*, para explicitar a repetição do padrão. As linhas 05 e 06 mostram a repetição do criptograma para o modo CBC. As linhas 10 e 11 mostram os criptogramas repetidos para o modo OFB. As repetições de criptograma para os modos CFB e CTR são mostradas nas linhas 14/15 e 18/19, respectivamente.

Um fato curioso dos encriptadores de bloco com comportamento de encriptador de fluxo (OFB, CFB e CTR) é que, para mesma chave e IV, o primeiro bloco do criptograma de cada um deles é idêntico ao primeiro bloco dos outros dois. Isto ocorre por que o criptograma do primeiro bloco é gerado simplesmente pelo XOR do texto claro com o IV encriptado com a chave, resultando sempre no mesmo valor quando há coincidência de parâmetros. Esta coincidência está marcada em vermelho na Saída 4.

Os modos de operação têm requisitos de geração e uso de IVs que precisam ser observados rigorosamente pelos programadores. Grosso modo, a boa prática dita que um IV nunca pode ser reutilizado com a mesma chave criptográfica no mesmo modo de operação. O modo CBC pode tolerar IVs pseudoaleatórios com pouquíssima chance de repetição. Os modos com comportamento de encriptadores de fluxo não toleram qualquer repetição de IV. A consequência da repetição é a revelação de mais informação do que padrões simples no criptograma.

Saída 4: IV fixo ou reutilizado por diversos modos de operação do AES.

```
01   Texto claro: Teste de IV fixoTeste de IV fixo
02   Chave      : 4A5349A8E49B4606746DC97BFB541384
03   IV fixo    : 0123456789ABCDEF0123456789ABCDEF
04
05   Encriptado com: AES/CBC/NoPadding
06   Criptograma1: 5064DB4A860AD38218AC16BE3E18344CBAD4FAE1E36AE256411E51577F5B7116
07   Criptograma2: 5064DB4A860AD38218AC16BE3E18344CBAD4FAE1E36AE256411E51577F5B7116
08
09   Encriptado com: AES/OFB/NoPadding
10   Criptograma1: 7ED81E3D99B68CC0CEB3CC19733087BFAB503D40DFE8F7F552D59356EC269EFC
11   Criptograma2: 7ED81E3D99B68CC0CEB3CC19733087BFAB503D40DFE8F7F552D59356EC269EFC
12
13   Encriptado com: AES/CFB/NoPadding
14   Criptograma1: 7ED81E3D99B68CC0CEB3CC19733087BF2D319CE0873444D1461EBD1B34952570
15   Criptograma2: 7ED81E3D99B68CC0CEB3CC19733087BF2D319CE0873444D1461EBD1B34952570
16
17   Encriptado com: AES/CTR/NoPadding
18   Criptograma1: 7ED81E3D99B68CC0CEB3CC19733087BFC9D6A25699F4877074723804A740086E
19   Criptograma2: 7ED81E3D99B68CC0CEB3CC19733087BFC9D6A25699F4877074723804A740086E
```

1.4.4. Sementes fixas ou reutilizadas para PRNGs

Já foi mostrado anteriormente na seção 1.3 que bons geradores de números pseudoaleatórios produzem sequências numéricas que se comportam, para a maioria dos usos práticos, como números aleatórios de fato. Porém, as sequências pseudoaleatórias são geradas por algoritmos determinísticos, os PRNGs, cujo fluxo de execução pode ser reproduzido a partir da repetição dos parâmetros de entrada. Quando PRNGs são utilizados na geração de IVs pseudoaleatórios, cuidados devem ser tomados para que não ocorra a repetição dos IVs por causa de uma configuração insegura como, por exemplo, o reuso de sementes fixas para o PRNG. Caso isto ocorra, um comportamento semelhante ao do reuso de IVs pode ser observado na geração dos criptogramas.

O Programa 10 ilustra a reutilização de IVs a partir da fixação de semente em um PRNG. O Programa 10 produz um resultado semelhante, mas não idêntico, ao da seção anterior, onde os IVs são reusados diretamente. As linhas 01 a 03 geram a chave AES de 128 bits. As linhas 04 e 05 criam um array com os quatro modos de operação CBC, OFB, CFB e CTR, sem *padding*. Para cada modo de operação, o laço iniciado na linha 06 faz o seguinte: cria duas instâncias (enc e dec) do AES no modo de operação em questão, cria um PRNG do tipo SHA1PRNG e o configura sempre com a mesma semente fixa, encripta o texto claro com a mesma chave e com o IV gerado internamente pelo PRNG, e decripta com a mesma chave e IV gerado pelo PRNG.

Visto que o SHA1PRNG tem na semente a única fonte de entropia para geração da sequência pseudoaleatória, a fixação da semente na linha 12 elimina a aleatoriedade da sequência, que será sempre a mesma. Este mau uso pode ser eliminado de duas maneiras: a primeira é simplesmente pela eliminação do comando *setSeed()* da linha 12, o que permitiria ao SHA1PRNG usar sementes diferentes em cada nova instância; a segunda é a substituição do SHA1PRNG por outro algoritmo que combine a semente com outras fontes de entropia, como é o caso do Windows-PRNG do sistema operacional Windows ou o /dev/urandom nos sistemas Linux. Uma terceira opção seria ainda combinar dois PRNGs, um para geração de semente e outro para geração da sequência pseudoaleatória.

Programa 10: Semente fixa ou reutilizada em PRNGs para geração de IVs.

```
01  KeyGenerator g = KeyGenerator.getInstance("AES", "BC");
02  g.init(128);
03  Key k = g.generateKey();
04  String[] aes = {"AES/CBC/NoPadding","AES/OFB/NoPadding",
05                  "AES/CFB/NoPadding","AES/CTR/NoPadding"};
06  for (int a = 0; a < aes.length; a++) {
07     Cipher enc = Cipher.getInstance(aes[a], "BC");
08     Cipher dec = Cipher.getInstance(aes[a], "BC");
09     byte[][] criptograma = new byte[2][];
10     for (int i = 0; i < 2; i++) {
11       SecureRandom sr = SecureRandom.getInstance("SHA1PRNG","SUN");
12       sr.setSeed(U.x2b("0123456789ABCDEF0123456789ABCDEF"));
13
14       enc.init(Cipher.ENCRYPT_MODE, k, sr);
15       criptograma[i] = enc.doFinal(textoClaroAna);
16
17       dec.init(Cipher.DECRYPT_MODE,k,new IvParameterSpec(enc.getIV()));
18       byte[] textoClaroBeto = dec.doFinal(criptograma[i]);
19  }}
```

1.4.5. Troca indevida de modos de operação

O mau uso por troca indevida de modos de operação é decorrente de dois mal-entendidos sobre a utilização de criptografia. O primeiro é a noção incorreta de que os modos de operação são intercambiáveis. Isto é, que um encriptador de fluxo como o AES/OFB poderia substituir (erroneamente), sem qualquer alteração no software criptográfico, um encriptador de bloco como o AES/CBC. O segundo mal-entendido está relacionado às consequências do reuso de chaves e IVs em encriptadores de fluxo.

Em um encriptador de fluxo, como AES/OFB, onde M é a mensagem em texto claro, C é o criptograma e o símbolo "^" é a operação lógica de OU-exclusivo (XOR), o fluxo de chaves K é obtido por uma cadeia de encriptações sucessivas do IV inicial. Neste encriptador de fluxo, obtém-se o criptograma como $K \wedge M = C$ e a decriptação como $C \wedge K = M$. No caso de dois criptogramas $C_1 = K_1 \wedge M$ e $C_2 = K_2 \wedge N$, obtém-se $C_1 \wedge C_2 = (K_1 \wedge M) \wedge (K_1 \wedge N) = M \wedge N$.

Se, por exemplo, as chaves são reusadas ($K_1 = K_2$) e o texto claro M possui partes fixas conhecidas, como é o caso das mensagens com cabeçalhos, então o texto claro N pode ser facilmente descoberto. A Saída 5 exemplifica a recuperação do texto claro N a partir da troca de um encriptador de bloco por um encriptador de fluxo em um sistema criptográfico que já estava mal configurado. Isto é, havia o reuso de IVs no modo CBC, cuja consequência é o reuso da chave no modo OFB. Na coluna da esquerda o texto claro é encriptado com AES/CBC, e na da direita com o AES/OFB.

No caso do CBC, a consequência do reuso de IVs para uma mesma chave pode ser menos prejudicial e geralmente pode passar despercebida por um longo tempo. Na coluna da esquerda da Saída 5, com modo CBC, a operação $C_0 \wedge C_1$ com M_0 conhecida não é capaz de revelar M_1 diretamente. Como pode ser observado pelo resultado da operação $C_0 \wedge C_1 \wedge M_0$ em caracteres não imprimíveis mostrados como "?". Por outro lado, na coluna da direita, com modo OFB, M_1 é revelada pela aplicação direta da operação $C_0 \wedge C_1 \wedge M_0$, que resulta em M_1. Este mau uso também pode ocorrer nos modos de operação CFB e CTR e encriptadores de fluxo verdadeiros como o RC4, podendo ser evitado pela utilização de um modo de encriptação autenticada, como o GCM com uma gestão rigorosa de IVs, conforme ilustrado na próxima seção.

Saída 5: Troca indevida de encriptador de bloco (AES/CBC) por um de fluxo (AES/OFB).

01	M[0] = "Troca a cifra de"	M[0] = "Troca a cifra de"
02	M[1] = "bloco por fluxo."	M[1] = "bloco por fluxo."
03		
04	K = 00112233445566778899AABBCCDDEEFF	K = 00112233445566778899AABBCCDDEEFF
05		
06	Encriptado com: AES/CBC/NoPadding	Encriptado com: AES/OFB/NoPadding
07		
08	C[0] =8C61DC63FF9682588910E6FF77866E58	C[0] =29CF058FFD0543D255888BC1A12F33DD
09		
10	iv[0]=BB3E5CFEDD9A5E6F666AC3ACB3D56D1C	iv[0]=BB3E5CFEDD9A5E6F666AC3ACB3D56D1C
11		
12	C[1] =00D0625EC980B1FD965931601CE279C3	C[1] =1FD1058FF305529D44C18BDFB5773896
13		
14	iv[1]=BB3E5CFEDD9A5E6F666AC3ACB3D56D1C	iv[1]=BB3E5CFEDD9A5E6F666AC3ACB3D56D1C
15		
16	C0^C1 = k^M0^K^M1 = M0^M1 =	C0^C1 = k^M0^K^M1 = M0^M1 =
17	=8CB1BE3D361633A51F49D79F6B64179B	=361E00000E00114F1149001E14580B4B
18		
19	M0^M1^M0 = M1 = ????????????????	M0^M1^M0 = M1 = bloco por fluxo.

1.4.6. Reutilização de chaves em encriptadores de fluxo

O reuso de chave em um encriptador de fluxo e a existência de partes fixas, como cabeçalhos, nos textos claros, podem viabilizar a descoberta de textos claros. Esta situação foi mostrada na seção anterior pelo reuso de IVs na geração do fluxo de chaves. Em um canal de comunicação bidirecional protegido por um encriptador de fluxo e uma chave secreta compartilhada, não basta que cada parte comunicante produza de modo único e imprevisível o seu IV. De fato, O IV deve ser único também no canal de comunicação e, por isto, combinado entre as partes para que não haja repetição. Em um caso recente [34], o mau uso de um encriptador de fluxo conhecido, o algoritmo RC4, em um protocolo de comunicação segura permitiu a decriptação de mensagens encriptadas trocadas entre dois usuários de um aplicativo de comunicação instantânea. A vulnerabilidade consistiu na reutilização do fluxo de chaves para encriptar mensagens nas duas direções do canal de comunicação protegido pelo encriptador de fluxo.

O trecho de código do Programa 11 mostra como Ivo pode descobrir o conteúdo de um criptograma de Beto a partir de uma mensagem indevidamente revelada por Ana em um sistema criptográfico mal configurado com repetição de IVs. As linhas de 01 a 07 contêm as configurações comuns a Ana e a Beto. Ana encripta nas linhas 09 e 10 e Beto encripta nas linhas 13 e 14. Ivo monitora a comunicação e lê os criptogramas de Ana e de Beto. Ivo (por exemplo, com engenharia social ou reversa) descobre o texto claro de Ana. Neste momento, Ivo usa as relações lógicas entre os criptogramas, a chave e os textos claros para recuperar o criptograma de Beto (linha 18).

Esta vulnerabilidade causada pelo mau uso do encriptador de fluxo pode ser evitada pela gestão compartilhada de IVs, em que cada parte comunicante gera o seu IV em um intervalo de valores diferente do utilizado pela outra parte, evitando deste modo a repetição do IV e o consequente reuso do fluxo de chaves. Por exemplo, em um IV de 128 bits, 1 bit pode identificar a parte comunicante (por exemplo, quem cria a chave usa o valor 1) e os 127 bits restantes podem ser incrementados por um contador. A chave deve ser trocada antes que qualquer lado atinja o limite do contador, evitando assim a repetição do IV para a mesma chave.

Programa 11: Reutilização de chaves em um encriptador de fluxo com AES/CTR.

```
01 // Configurações comuns para Ana e Beto
02 byte[][] M = {("Reuso de chave d").getBytes(),
03               ("a Cifra de fluxo").getBytes()};
04 byte[][] iv = {U.x2b("0123456789ABCDEF0123456789ABCDEF"),
05                U.x2b("0123456789ABCDEF0123456789ABCDEF")};
06 SecretKeySpec ks = new SecretKeySpec(k, "AES");
07 Cipher enc = Cipher.getInstance("AES/CTR/NoPadding", "BC");
08
09 // Ana encripta
10 enc.init(Cipher.ENCRYPT_MODE,ks,new IvParameterSpec(iv[0]));
11 C[0] = enc.doFinal(M[0]);
12
13 //Beto Encripta também
14 enc.init(Cipher.ENCRYPT_MODE, ks, new IvParameterSpec(iv[1]));
15 C[1] = enc.doFinal(M[1]);
16
17 // Ivo realiza o ataque
18 byte[] M0xorM1 = U.xor(C[0],C[1]); byte[] M1 = U.xor(M[0], M0xorM1);
```

```
01 //C0^C1 = k^M0^K^M1 = M0^M1 = 3345361A09520545440648071A10580B
02 //Resultado: M0^M1^M0 = M1 = a Cifra de fluxo
```

1.4.7. Maleabilidade indevida dos encriptadores de fluxo

Os criptogramas produzidos por sistemas criptográficos exclusivamente para sigilo, sem mecanismos de verificação de integridade e autenticidade, não possuem garantias de integridade e estão sujeitos a corrupções acidentais ou maliciosas. Em particular, os criptogramas gerados pelos encriptadores de fluxo são maleáveis e podem ser intencionalmente modificados de modo a produzir alterações controladas no texto claro.

Seja $C = M\hat{}K$ o criptograma de Ana obtido pelo XOR do fluxo de chaves K com o texto claro M. Aqui K foi bem construído e não é reutilizado. Ivo não precisa conhecer K. Porém, M possui estrutura e partes do conteúdo conhecidas por Ivo e, por isto, ele consegue realizar com precisão as modificações desejadas. Ivo deseja alterar M para o valor N a partir de mudanças controladas sobre C. Ivo calcula a mudança $X = M\hat{}N$. Ivo intercepta C em trânsito, calcula o criptograma modificado $Y = C\hat{}X$ e envia Y para Beto. Beto recebe Y e calcula $Y\hat{}K = (C\hat{}X)\hat{}K = (M\hat{}K)\hat{}(M\hat{}N)\hat{}K = (M\hat{}M)\hat{}(K\hat{}K)\hat{}N = 1\hat{}1\hat{}N = N$. Desta forma, Beto obtém apenas o texto claro falsificado N.

O trecho de código do Programa 12 mostra como criptogramas produzidos com o AES/CTR podem ser modificados, resultando em textos claros falsos, porém de conteúdo válido no contexto da aplicação. Ana e Beto usam o mesmo encriptador de fluxo da linha 02. A mensagem de Ana para Beto é composta de dois blocos de texto claro, na linha 05, em que Ana faz uma transferência de valor para Carlo. Nas linhas de 07 a 10 os blocos do criptograma são gerados conforme pretendido por Ana e Beto. Vale dizer que o IV do segundo bloco é incrementado de um, para evitar a repetição.

Nas linhas 14 a 17, Ivo realiza seu ataque. Primeiro, ele inclui seu nome na mensagem do seguinte modo. Nas linhas 14 e 15, Ivo calcula X = *"Carlo"^" Ivo"* e $C_0 = C_0\hat{}X$. Em seguida, nas linhas 16 e 17, Ivo aumenta o valor da mensagem calculando X = *"010.000,00"^" 100.998,54"* e, em seguida, $C_1 = C_1\hat{}X$. O resultado recebido por Beto são os blocos da mensagem falsa *"Ana para Ivo"* e *"Valor:100.998,54"*.

Programa 12: Maleabilidade dos encriptadores de fluxo com AES/CTR.

```
01 // Configurações comuns para Ana e Beto
02 Cipher c = Cipher.getInstance("AES/CTR/NoPadding", "BC");
03
04 // A mensagem é composta de dois blocos de texto claro
05 byte[][]M = {("Ana   para Carlo").getBytes(),
06              ("Valor:010.000,00").getBytes()};
07
08 for (int i= 0; i < M.length; i++){
09   c.init(Cipher.ENCRYPT_MODE,ks,new IvParameterSpec(iv[i]));
10   C[i] = c.doFinal(M[i]);
11 }
12
13 //Ivo realiza o ataque
14 X = U.xor("Ana   para Carlo".getBytes(),
15           "Ana   para   Ivo".getBytes()); C[0] = U.xor(C[0], X);
16 X = U.xor("Valor:010.000,00".getBytes(),
17           "Valor:100.998,54".getBytes()); C[1] = U.xor(C[1], X);
```

```
01 // Resultado: Ivo é o novo recebedor de um valor muito mais alto
02 // Resultado: N[0] =Ana   para   Ivo
03 // Resultado: N[1] =Valor:100.998,54
```

1.4.8. Combinação de integridade e encriptação

O mau uso descrito na seção anterior não pode ser evitado somente pela verificação de integridade da mensagem. De fato, *hashes* não são capazes de detectar a corrupção maliciosa de mensagens maleáveis. Esta ideia incorreta constitui outro mau uso da criptografia. O trecho de código do Programa 13 mostra como a combinação de encriptação com verificação de integridade por *hash* não é suficiente para proteger um criptograma maleável contra modificações intencionais e maliciosas. As linhas 01 a 04 contêm as configurações do sistema criptográfico para Ana e Beto. Neste exemplo vale ressaltar a função de resumo criptográfico SHA256 na linha 03. A encriptação é feita com AES/CTR.

Nas linhas 06 a 09, Ana inicializa as funções de *hash* e de encriptação, computa o criptograma do texto claro e calcula o *hash* do criptograma. Esta técnica é conhecida como *Encrypt-then-Hash* (EtH). Nas linhas 11 a 15, Ivo intercepta a mensagem de Ana para Beto, e se aproveita da maleabilidade do encriptador de fluxo para modificar o criptograma, de modo a incluir seu próprio nome no lugar do nome de Beto no texto claro. Além disso, Ivo recalcula o *hash* sobre o novo criptograma e o envia, junto com o novo criptograma, para Beto.

Nas linhas 17 a 20, ao receber a mensagem com o criptograma e o *hash*, Beto inicializa as funções de *hash* e de encriptação, calcula o *hash* sobre o criptograma e compara o *hash* calculado com o *hash* recebido. Uma vez que estes dois valores são iguais, Beto decripta o criptograma e obtém o texto claro. Neste exemplo, Beto foi incapaz de detectar a modificação maliciosa do texto claro feita por Ivo, uma vez que Ivo também substituiu o *hash* de verificação de integridade. Felizmente este mau uso é facilmente corrigido pela utilização de encriptação autenticada, em que a função de encriptação pode ser combinada com uma função de MAC, como por exemplo, o HMACSHA256, ou então substituída por uma única função de encriptação autenticada, como por exemplo, o AES/GCM.

Programa 13: Combinação de interidade e encriptação com SHA256 e AES/CTR.

```
01  // configurações do sistema criptográfico para Ana e Beto
02  Cipher c = Cipher.getInstance("AES/CTR/NoPadding", "BC");
03  MessageDigest md = MessageDigest.getInstance("SHA256", "BC");
04  byte[] textoclaroAna = "De Ana para Beto".getBytes();
05
06  //Ana calcula o hash do criptograma: Encrypt-then-Hash(EtH)
07  md.reset(); c.init(Cipher.ENCRYPT_MODE,sks,ivps);
08  byte[] criptograma = c.doFinal(textoclaroAna);
09  byte[] hash = md.digest(criptograma);
10
11  //Ataque: Ivo modifica o criptograma e recalcula o hash
12  X = U.xor("De Ana para Beto".getBytes(),
13            "De Ana para  Ivo".getBytes());
14  criptograma = U.xor(criptograma, X);
15  hash = md.digest(criptograma);
16
17  // decriptação pelo Beto com verificação da hash
18  md.reset(); c.init(Cipher.DECRYPT_MODE,sks,ivps);
19  if (MessageDigest.isEqual(md.digest(criptograma), hash))
20     { textoclaroBeto = c.doFinal(criptograma);}
```

1.4.9. Combinação manual de MAC e encriptação

A encriptação autenticada pode ser obtida manualmente pela combinação programática das funções de encriptação e autenticação. Em geral, existem três métodos para fazer a combinação das funções de MAC e encriptação, que se diferenciam em relação ao dado sobre o qual a *tag* de autenticação é calculada, conforme a seguir:

- Encrypt-then-MAC (EtM): encripta então autentica. Este método calcula a *tag* de autenticação sobre o criptograma e a *tag* não é encriptada;
- Encrypt-and-MAC (E&M): encripta e autentica. Este método calcula a *tag* de autenticação sobre o texto claro e a encriptação é feita somente sobre o texto claro e a *tag* não é encriptada;
- MAC-then-Encrypt (MtE): autentica então encripta. Este método calcula a *tag* de autenticação sobre o texto claro e depois encripta o texto claro e a *tag*.

O primeiro método, encripta então autentica, é considerado o mais seguro e foi exemplificado na seção anterior. Uma das vantagens do método EtM é que ele não revela informação sobre a integridade de formação do texto claro, uma vez que a *tag* foi calculada sobre o criptograma. Por isto, este método não é vulnerável ao ataque de *padding oracle*. A combinação manual pode ser substituída por uma única função de encriptação autenticada, como o AES/GCM. O trecho de código do Programa 14 mostra os outros dois métodos: encripta e autentica (E&M) e autentica então encripta (MtE). As configurações do sistema criptográfico estão nas linhas de 01 a 05. Duas chaves criptográficas são necessárias, uma para o AES/CTR e outra para o HMACSHA256.

O método encripta e autentica é mostrado nas linhas de 07 a 11. O criptograma é computado na linha 10 e a *tag* é calculada sobre o texto claro na linha 11. A verificação da *tag* requer a decriptação do criptograma. O método autentica então encripta é mostrado nas linhas de 13 a 18. A *tag* é calculada sobre o texto claro na linha 16, a *tag* é concatenada ao final do texto claro, linha 17, formando o pacote sobre o qual o criptograma é computado, na linha 18. A verificação da *tag* requer a decriptação do criptograma e a separação entre texto claro e *tag*. Nestes dois métodos, a verificação de integridade do texto claro é determinística e revela a repetição de um texto claro.

Programa 14: Encriptação autenticada manual com Encrypt-and-MAC e MAC-then-Encrypt.

```
01  //configurações do sistema criptográfico
02  SecretKeySpec sks1 = new SecretKeySpec(k, "AES");
03  SecretKeySpec sks2 = new SecretKeySpec(k, "HMACSHA256");
04  Cipher c = Cipher.getInstance("AES/CTR/NoPadding", "BC");
05  Mac m = Mac.getInstance("HMACSHA256", "BC");
06
07  // encripta e autentica: Encrypt-and-MAC (E&M)
08  m.init(sks2);
09  c.init(Cipher.ENCRYPT_MODE, sks1, new IvParameterSpec(iv));
10  criptograma = c.doFinal(textoclaroAna);
11  tag = m.doFinal(textoclaroAna);
12
13  // autentica entao encripta: MAC-then-Encrypt (MtE)
14  m.init(sks2);
15  c.init(Cipher.ENCRYPT_MODE, sks1, new IvParameterSpec(iv));
16  tag = m.doFinal(textoclaroAna);
17  byte[] pacote = Arrays.concatenate(textoclaroAna,tag);
18  criptograma = c.doFinal(pacote);
```

1.4.10. Tempo variável na verificação de *hashes* e MACs

Os maus usos criptográficos podem levar a vazamentos de informação por canais laterais (*side-channels* em inglês). Um canal lateral (de tempo) é alimentado pelas variações de tempo das computações sobre valores criptográficos. Por exemplo, quando a comparação de *hashes* ocorre em tempo variável, as pequenas variações de tempo na comparação revelam onde está a diferença entre valores comparados e podem ser utilizadas por Ivo para deduzir o valor de *hash* original. Para evitar este vazamento de informação, a comparação de *hashes* e MACs deve ser realizada em tempo constante e independente de onde ocorre a primeira diferença entre os valores comparados.

O trecho de código do Programa 15 mostra como é possível medir a diferença de tempo de comparação entre dois *hashes* de 64 bytes do SHA512. O laço da linha 05 percorre os 64 bytes do *hash* calculado, que é modificado no índice i (na linha 08). A comparação é feita pelo método Arrays.areEqual() que tem tempo variável. O resultado da tomada de tempo é mostrado na Figura 17, onde se percebe que o tempo de comparação aumenta com o índice i da primeira diferença entre valores comparados. Já os métodos MessageDigest.isEqual() e Arrays.ConstTimeAreEqual() oferecem tempos constantes, com vantagem de desempenho para o primeiro.

Programa 15: Tempo variável na verificação de hashes com SHA512.

```
01  MessageDigest md = MessageDigest.getInstance("SHA512", "BC");
02  boolean ok; long t1, t2; long t[] = new long[64];
03  byte[] hash1 = md.digest(textoClaro.getBytes());
04
05  for (int i = 0; i < t.length; i++) { // 64 bytes
06    md.reset();
07    byte[] hash2 = md.digest(textoClaro.getBytes());
08    hash2[i] = (byte) (hash2[i] ^ 0x01);
09    t1 = System.nanoTime();
10    ok = Arrays.areEqual(hash2, hash1);
11    t2 = System.nanoTime();
12    t[i] = t2 - t1;
13  }
```

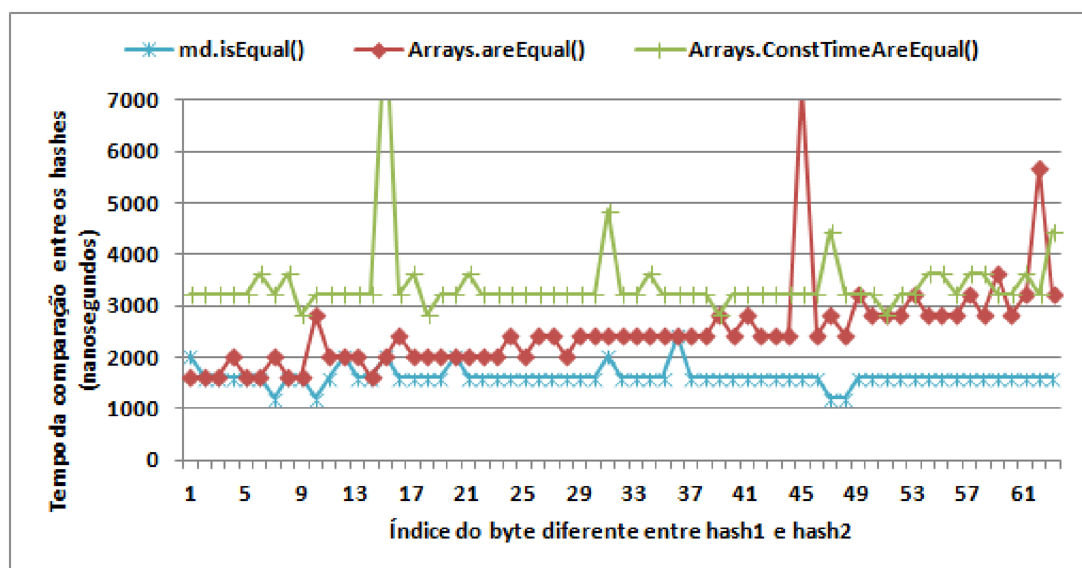

Figura 17: Comparação de *hashes* com tempo variável e com tempo constante.

1.4.11. *Padding oracles* do modo CBC

A vulnerabilidade de *padding oracle* (oráculo de preenchimento, em uma tradução livre) do modo CBC é um tipo de canal lateral conhecido pelos criptólogos desde 2002 [35], mas que só recentemente foi explorado em ataques reais a sistemas web [33][36]. Grosso modo, o *padding oracle* é um algoritmo que, ao receber um criptograma qualquer como entrada, retorna um único bit de informação sobre a validade do *padding* que completa o texto claro correspondente. Isto é, se o *padding* é válido, o oráculo retorna 1 ou *true* (verdadeiro); se ele não é válido, o oráculo retorna 0 ou *false* (falso). Ao usar estas respostas sobre a validade do texto claro, um adversário pode deduzir valores retornados pela função de decriptação.

Chama-se de vulnerabilidade do *padding oracle* a este vazamento de um bit da função de decriptação que, ao tentar decriptar qualquer criptograma, emitirá uma mensagem de erro informando quando o *padding* é inválido. Este vazamento de um bit pode ser utilizado iterativamente para descobrir todos os bytes de um texto claro, geralmente ao custo de apenas alguns milhares de iterações. Geralmente, a função de decriptação é executada mesmo que não retorne o texto claro para o adversário. Entretanto, ela deve informar se o texto claro tem um *padding* válido. Em situações extremas, o *padding oracle* serve não apenas como meio para ataques de decriptação de mensagens, mas também como bloco de construção para oráculos de encriptação.

O trecho de código do Programa 16 exemplifica um *padding oracle* simples. O oráculo recebe como entrada dois blocos de dados, o *iv* e o criptograma *c* (linha 01) e retorna *true* se a decriptação for bem sucedida. A decriptação de *c* é realizada (linha 06), mas não é retornada. Na linha 07, o oráculo trata a exceção de preenchimento ruim (`BadPaddingException`) do texto claro e retorna o valor *false* se ela ocorrer. Para este exemplo, todas as outras exceções são ignoradas. O valor lógico retornado na linha 12 é suficiente para a realização do ataque.

O diagrama de sequência da Figura 18 ilustra a implementação do ataque de *padding oracle* do modo CBC. Primeiro, o atacante Ivo obtém um criptograma e um IV conhecido e, iterativamente, modifica o byte menos significativo do IV e submete IV e *c* ao oráculo para descobrir o valor do *padding*. Em seguida, Ivo, mais uma vez pela manipulação do IV, modifica iterativamente o *padding* descoberto para decriptar, byte a byte, o criptograma, fazendo questionamentos repetidos ao oráculo. *Padding oracles* como este já foram encontrados em diversos softwares criptográficos [33][36].

Programa 16: *Padding oracle* do modo CBC.

```
01  public static boolean oracle(byte[] iv, byte[] c) {
02    boolean ok = true;
03    try {
04      Cipher c = Cipher.getInstance("AES/CBC/PKCS7Padding", "BC");
05      c.init(Cipher.DECRYPT_MODE, ks, new IvParameterSpec(iv));
06      c.doFinal(c); // ignora a saída do doFinal()!!!!
07    } catch (BadPaddingException e) { ok = false;
08    } catch (NoSuchAlgorithmException | NoSuchProviderException |
09             NoSuchPaddingException | InvalidKeyException |
10             InvalidAlgorithmParameterException |
11             IllegalBlockSizeException ex) { /* ignora!! */}
12  return ok;}
```

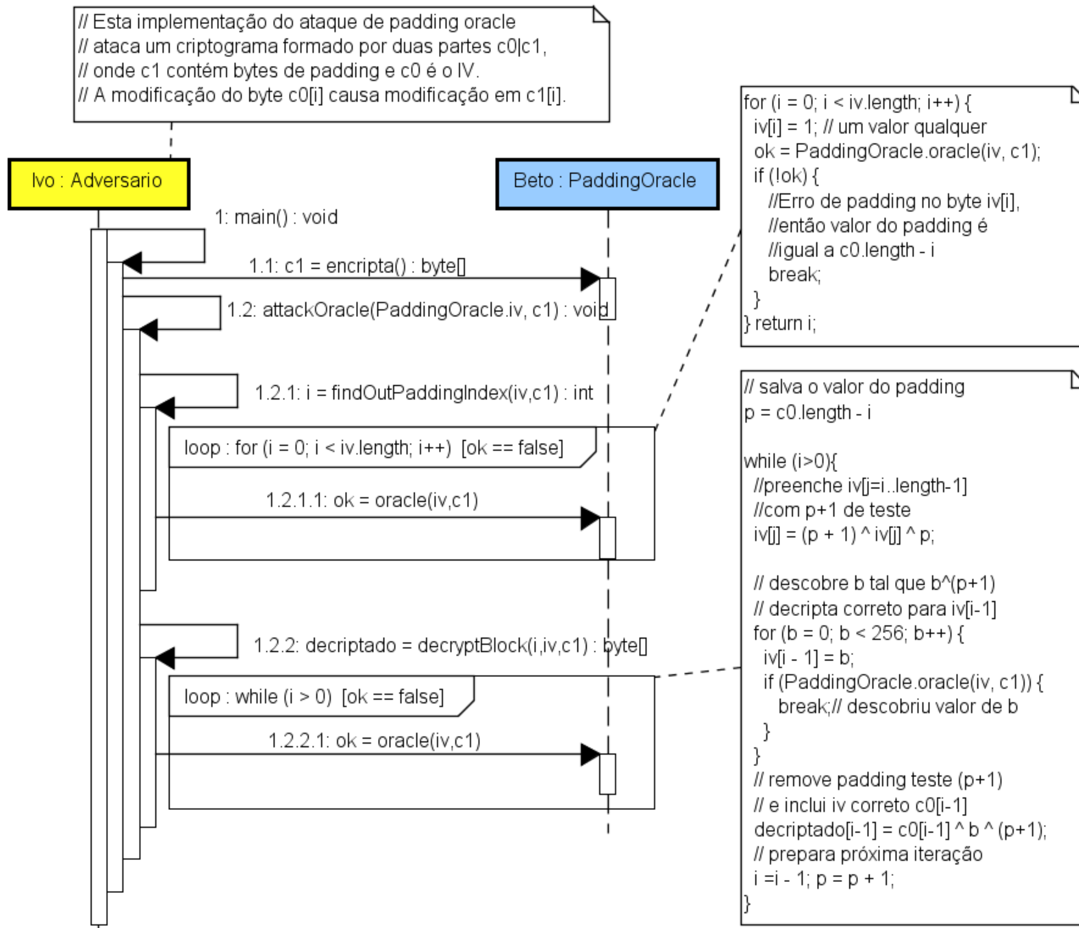

Figura 18: Diagrama de sequência (UML) do ataque de *padding oracle* do modo CBC.

Existem três maus usos criptográficos que podem ser diretamente associados às causas da vulnerabilidade de *padding oracle*:

- O uso de encriptação não autenticada em canais de comunicação, de modo que criptogramas injetados pelo adversário no canal sejam considerados válidos;
- Composição de encriptação e autenticação com os métodos MAC-then-Encrypt ou MAC-then-(pad-then)-Encrypt, de modo que a proteção de integridade só é aplicada sobre o texto claro (e o *padding*) e não sobre o criptograma, permitindo ao adversário a manipulação do criptograma para inferir a validade do *padding*;
- O uso de IVs fixos ou previsíveis com o modo CBC, de modo que ataques de decriptação possam ser montados a partir da previsão dos IVs.

Estes maus usos podem ser resolvidos pela adoção compulsória da autenticação em conjunto com a encriptação, que também protege a integridade. Ainda, ela precisa ser aplicada de maneira correta, sobre o criptograma completo, em vez de sobre o texto claro. Além disso, a função atômica de encriptação autenticada elimina a necessidade de combinação programática explícita de encriptação e autenticação, reduzindo a chance de erros. Finalmente, a última contramedida para eliminar este canal lateral é a inibição de mensagens de erro relacionadas à validação de integridade sobre o texto claro nas computações que fazem decriptação em protocolos de comunicação.

1.4.12. Validação incompleta de certificados digitais

A validação de certificados digitais em softwares criptográficos é relativamente bem estruturada e confiável, uma vez que as falhas de programação mais comuns no uso de certificados digitais já foram identificadas e solucionadas. Conforme discutido anteriormente, os quatro pontos mais importantes na validação do certificado digital são a confirmação do nome do sujeito ou usuário, a verificação da assinatura digital presente no certificado (e a consequente verificação da cadeia de certificação), a verificação da data de expiração e a presença do número do certificado em uma lista de revogação.

Em softwares criptográficos em geral, e nos aplicativos móveis em especial, existe uma variedade grande de bibliotecas para estabelecimento de conexões SSL/TLS. De acordo com estudos recentes [16][19], todas elas permitem que o programador desabilite um ou outro item na verificação do certificado, para fins de desenvolvimento. Porém, há risco da configuração desabilitada ser implantada acidentalmente em ambiente de produção. Em particular, a não verificação da assinatura ou da cadeia de certificação facilita o ataque do intermediário malicioso (*Man-in-The-Middle* – MiTM).

A vulnerabilidade Heartbleed [31] descoberta no código da extensão do TLS em abril de 2014 e presente no software criptográfico OpenSSL [40] é um exemplo de como defeitos comuns podem resultar em vulnerabilidades graves nos softwares criptográficos. A vulnerabilidade foi causada pela falta de verificação dos limites em um buffer em que o *nonce* é armazenado pelo protocolo Heartbeat. Na linguagem C, o tamanho do buffer é desvinculado da variável que registra este seu tamanho. A falta de consistência entre tamanho do buffer e a variável que registra o tamanho pode ser explorada por meio de leitura estendida de posições de memória além do tamanho real do buffer. Este defeito de programação insegura simples, quando presente em um protocolo de comunicação, como foi o caso do protocolo Heartbeat, possibilitou a leitura de regiões de memória do processo do OpenSSL, no servidor, em que eram mantidas diversas informações sensíveis tais como chaves de sessão, senhas de usuário e até chaves privadas de servidores, facilitando os ataques MiTM.

Outra vulnerabilidade recente [32] é mais um defeito do OpenSSL descoberto no sistema operacional iOS da Apple em fevereiro de 2014. O defeito consistiu de duas sentenças “goto fail” consecutivas e sem chaves delimitadoras colocadas depois de uma cláusula condicional (“if”). Este código resultou em um salto incondicional para a sentença rotulada de “fail”. Este defeito ocorreu em um ponto crítico do estabelecimento de sessão do SSL, na verificação de uma assinatura digital; por causa do defeito, passou a ser considerada sempre bem sucedida, mesmo na ocorrência de certificados inválidos ou assinatura inexistente, mais uma vez favorecendo o ataque MiTM.

Estas vulnerabilidades poderiam ter sido facilmente identificadas pela aplicação de técnicas comuns de auditoria de código fonte, manuais ou até automáticas. Por outro lado, as vulnerabilidades poderiam nunca ter ocorrido se linguagens de programação com tipos fortes tivessem sido utilizadas. Para ilustração, o software de infraestrutura de chaves públicas EJBCA [37], escrito em Java e que utiliza à biblioteca criptográfica BouncyCastle, também em Java, são imunes às vulnerabilidades de leitura de buffer além dos limites, que causaram a vulnerabilidade Heartbleed.

1.5. Por dentro de um encriptador de blocos

Este seção ilustra a construção de uma implementação do algoritmo AES conforme a especificação [23], em que são destacados os aspectos necessários para utilizá-la como um encriptador de blocos [26]. Além disso, são tratadas otimizações algorítmicas e de código, automação de testes funcionais [24] e análise de segurança da implementação.

1.5.1. O algoritmo AES

A estrutura de dados principal do AES é o *square*, uma matriz 4x4 de células de 1 byte cada, totalizando 128 bits. Todas as operações do AES são realizadas sobre esta matriz. Computações também são feitas sobre palavras de quatro bytes. Uma palavra é uma linha ou uma coluna da matriz. O conjunto de valores correntes da matriz é chamado de estado. Cada rodada do AES usa uma subchave de 128 bits derivada da chave principal por meio de um mecanismo de expansão de chaves. O AES trabalha com três tamanhos de chaves: 128, 192 e 256 bits. O laço principal do AES possui 10,12 ou 14 rodadas, conforme o tamanho da chave. O Programa 17 mostra a função de encriptação do AES, com as operações sobre a estrutura de estado: adição da chave da rodada, a substituição de bytes, a rotação de linhas, e a mistura de colunas. A decriptação é análoga, com as operações inversas.

1.5.2. Modos de operação e *padding*

A implementação crua do AES é de pouca utilidade prática. A fim de torná-la útil é necessário acrescentar as construções necessárias para torná-la um encriptador de bloco: modos de operação e esquema de *padding* do texto claro, para que este tenha o tamanho de um múltiplo do tamanho do bloco do algoritmo de encriptação. Em um encriptador de bloco, a operação de encriptação insere o *padding* e em seguida encripta com o modo de operação determinado. A operação de decriptação decripta com o modo de operação correspondente e em seguida remove o *padding*, conforme visto no Programa 18 (linhas 01 a 15). As linhas 17 a 55 mostram os modos de operação ECB (17 a 25), CBC (27 a 41) e CTR (43 a 54) [26]. O *padding* PKCS#5/PKCS#7 [21] está a partir da linha 58.

Programa 17: Função de encriptação do AES.

```
01  public void encrypt(byte[] in, byte[] out) {
02    wCount = 0;
03    byte[][] state = new byte[4][Nb];
04    Util.toState(state, in);
05    addRoundKey(state);
06    for (int round = 1; round < Nr; round++) {
07      subBytes(state);
08      shiftRows(state);
09      mixColumns(state);
10      addRoundKey(state);
11    }
12    subBytes(state);
13    shiftRows(state);
14    addRoundKey(state);
15    Util.fromState(out, state);
16  }
```

Programa 18: Modos de operação e *padding* de um encriptador de bloco.

```
01  public byte[] doCipher(byte[] in){
02    byte[] inPadded = null, outPadded = null, out = null;
03    if (padding){
04      if(enc==ENCRYPT) {inPadded = doPadding(in);}
05      if(enc==DECRYPT) {inPadded = in;}
06    } else { inPadded = in;}
07    outPadded = new byte[inPadded.length];
08          if (this.opMode == ECB) {doECB(inPadded,outPadded);}
09    else if (this.opMode == CBC) {doCBC(inPadded,outPadded);}
10    else if (this.opMode == CTR) {doCTR(inPadded,outPadded);}
11    if (padding){
12      if (enc == ENCRYPT) { out = outPadded; }
13      if (enc == DECRYPT) { out = undoPadding(outPadded);}
14    } else { out = outPadded;} return out;
15  }
16
17  private void doECB(byte[] in, byte[] out) {
18    int numBlocks = (int)(in.length/blockSize);
19    for (int i = 0; i < numBlocks; i++) {
20      int offset = i * blockSize;
21      for (int j=0; j<blockSize; j++) { inBlock[j] = in[offset+j];}
22      if      (enc == ENCRYPT) { aes.encrypt(inBlock, outBlock);}
23      else if (enc == DECRYPT) { aes.decrypt(inBlock, outBlock);}
24      for (int j=0; j<blockSize; j++) {out[offset+j] = outBlock[j];}
25  }}
26
27  private void doCBC(byte[] in, byte[] out) {
28    int numBlocks = (int)(in.length/blockSize);
29    byte[] inX = new byte[blockSize], outX = new byte[blockSize];
30    for (int i=0; i<numBlocks; i++) {
31      int offset = i * blockSize;
32      for (int j=0; j<blockSize; j++) {inBlock[j] = in[offset+j];}
33      if (i == 0) {copy(iv,outX);}
34      if (enc == ENCRYPT) {
35        xor(inBlock,outX,inX);
36        aes.encrypt(inX,outBlock);
37        copy(outBlock,outX);}
38      if (enc == DECRYPT) {
39        aes.decrypt(inBlock, inX);
40        xor(inX,outX,outBlock);
41        copy(inBlock,outX)   ;}
42      for (int j=0; j<blockSize; j++) {out[offset+j] = outBlock[j];}
43  }}
44
45  private void doCTR(byte[] in, byte[] out) {
46    int numBlocks = (int)(in.length/blockSize), sizeCTR = blockSize/2;
47    byte[] inCTR = new byte[blockSize], outCTR = new byte[blockSize];
48    for (int i = 0; i < numBlocks; i++) {
49      int offset = i * blockSize;
50      for (int j=0; j<blockSize; j++) {inBlock[j] = in[offset+j];}
51      if (i == 0) {copy(iv,inCTR);}
52      else {standardIncrement(inCTR,sizeCTR);}
53      if (enc == ENCRYPT || enc == DECRYPT)
54      { aes.encrypt(inCTR, outCTR); xor(outCTR,inBlock,outBlock);}
55      for(int j=0; j<blockSize; j++) {out[offset+j] = outBlock[j];}
56  }}
57
58  private byte[] doPadding(byte[] in){
59    byte pad[] = {16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1};
60    byte val   = (byte) (in.length%blockSize);
61    int numBlocks = (int)(in.length/blockSize);
62    byte[] padded = new byte[(numBlocks+1)*blockSize];
63    for(int i=0; i<in.length; i++){padded[i] = in[i];}
64    for(int i=in.length; i<padded.length; i++){padded[i]=(byte)pad[val];}
65    return padded;
66  }
67  private byte[] undoPadding(byte[] padded){
68    byte[] out = new byte[padded.length - padded[padded.length-1]];
69    for(int i=0; i<out.length; i++){ out[i]=padded[i];} return out;}
```

Programa 19: Dois métodos de multiplicação.

```
01   public byte FFMul(byte a, byte b) {
02     byte aa = a, bb = b, r = 0, t;
03     while (aa!=0){
04       if ((aa&1) != 0) {r = (byte)(r^bb);}
05       t = (byte) (bb&0x80); bb =(byte)(bb << 1);
06       if (t != 0) {bb =(byte) (bb^0x1b);}
07       aa =(byte)((aa & 0xff) >> 1);
08     }
09     return r;
10   }
11   public byte FFMulFast(byte a, byte b) {
12    int t = 0;
13    if (a == 0 || b == 0) { return 0;}
14    t = (L[(a & 0xff)] & 0xff) + (L[(b & 0xff)] & 0xff);
15    if (t > 255) { t = t - 255;}
16    return E[(t & 0xff)];
17   }
```

1.5.3. Otimização da implementação do algoritmo AES

A implementação do AES sofreu otimizações algorítmicas e de código. A substituição do método de multiplicação utilizado pelas operações de encriptação e decriptação foi a que resultou no maior ganho de desempenho, pois baixou a complexidade do algoritmo. O método convencional, com deslocamentos de bits, foi substituído pelo método mais eficiente baseado em *lookup* de tabelas. O Programa 19 mostra os dois métodos, o mais lento (linhas 1 a 10) e o mais rápido (linhas 11 a 17).

Outras otimizações realizadas foram a desenrolagem de laços dentro do fluxo de controle do algoritmo e a pré-computação das tabelas. Nestas otimizações de código os ganhos foram mais modestos. A Figura 19 mostra o ganho de desempenho acumulado com as otimizações combinadas: sem otimização, otimização algorítmica (multiplicação rápida) e otimizações de código. A figura mostra os tempos, em milissegundos, de encriptação e decriptação para três implementações, nos três tamanhos de chave. A otimização algorítmica ofereceu a maior contribuição no desempenho final. A decriptação é mais lenta por causa da inversão modular na decriptação.

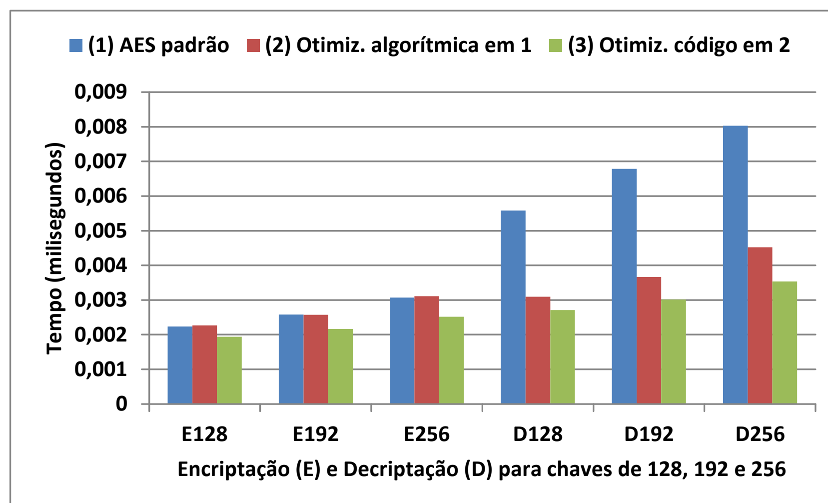

Figura 19: Tempos de encriptação e decriptação para três implementações do AES.

1.5.4. Validação funcional com vetores de teste

A validação funcional de implementações criptográficas pode ser realizada com o auxílio de vetores de teste criptográficos, que são conjuntos de dados construídos com o objetivo de avaliar a correção de implementações criptográficas em relação às normas e especificações. A correção funcional de uma implementação criptográfica é premissa para a segurança desta implementação, uma vez que uma implementação incorreta não oferece segurança. Porém, a validação não oferece uma medida da segurança da implementação. Além de ajudar a determinar a conformidade com as especificações, a validação pode detectar falhas de implementação, incluindo problemas com ponteiros, a alocação insuficiente de memória, tratamento incorreto de erros e comportamentos anômalos diversos na implementação avaliada.

Devido ao grande volume de dados envolvidos, os vetores de teste são utilizados em testes automáticos das implementações criptográficas. Para que a validação seja viável, o software criptográfico deve permitir ao software de validação ter o controle sobre os parâmetros de entrada necessários aos testes, por exemplo, por meio de uma interface de programação (API). Se um software criptográfico não permite o controle dos parâmetros de entrada, os testes não podem ser realizados satisfatoriamente.

Existem diversas bases ou vetores de teste para validação de implementações de algoritmos criptográficos. Um conjunto de vetores de boa reputação é o do NIST CAVP [24]. Os vetores de testes devem ter uma amostragem estatística, senão exaustiva, que permita pelo menos a cobertura completa dos fluxos de controle do algoritmo. Quanto melhor a cobertura da validação, maior será a confiança na correção da implementação. Além disto, a validação de implementações contra amostras diversas e atualizadas é uma boa prática de teste de software. A Figura 20 conceitua a validação de algoritmos com vetores de teste criptográficos. Vale observar que em se tratando de algoritmos não padronizados, geralmente o próprio criptólogo disponibiliza os vetores de teste.

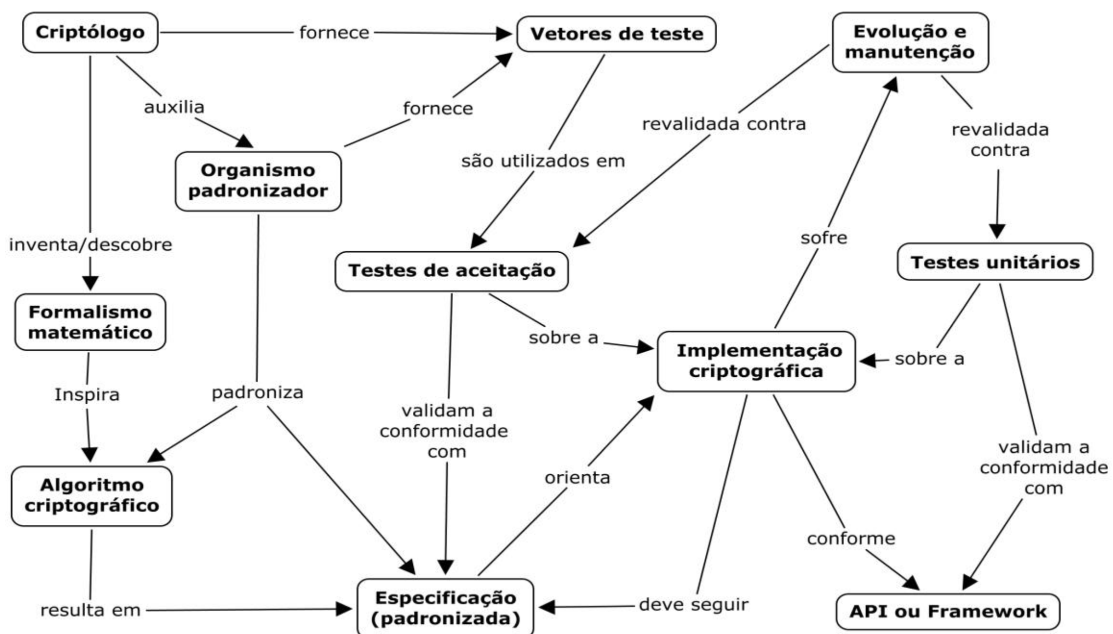

Figura 20: O conceito de validação de algoritmos criptográficos com vetores de teste.

1.5.5. Análise de segurança da implementação

Neste ponto, supõe-se que a implementação tenha passado pelos testes de validação e possa ser considerada, em certa medida, aderente à sua especificação com uma confiança justificada pela cobertura ou abrangência da validação. Isto, porém, não significa que a implementação seja segura. Em particular, a implementação está sujeita às vulnerabilidades dos sistemas criptográficos e de computação em que será utilizada.

Conforme visto anteriormente, existem diversas maneiras de usar mal um encriptador de blocos. Por exemplo, o modo ECB é determinístico e está sujeito à repetição de padrões, enquanto o modo CBC está sujeito ao mau uso de IVs e, combinado ao *padding* PKCS#7, é vulnerável ao ataque por canal lateral conhecido como *padding oracle*. Já o modo CTR é seriamente afetado pelo reuso de IVs.

Além de variações do tempo de execução, ataques por canais laterais também usam monitoramento do consumo de energia, uso da memória, emanação eletromagnética, tratamento de erros, entre outras [5]. Como já vimos, as diferenças de tempo durante as computações com chaves criptográficas podem levar a ataques de canal lateral, assim como as diferenças de tempo entre o processamento bem sucedido e o tratamento de erros e exceções. Uma implementação criptográfica deve evitar os vazamentos por canais laterais. Entretanto, nem sempre é possível obter uma solução efetiva somente em software, sem o apoio do hardware subjacente ou das outras camadas de software intermediárias.

Em Java, as modificações no código fonte feitas para eliminar canais laterais podem ser desfeitas pela JVM durante o processo de compilação em tempo real (*Just-in-Time Compilation* - JiTC). Este processo pode não apenas desfazer proteções, como também acrescentar vulnerabilidades relacionadas aos canais laterais. Este é o caso das construções em código fonte para computações em tempo constante.

Finalmente, implementações do AES são particularmente vulneráveis aos ataques por canais laterais sobre os acessos à memória *cache* dos processadores [5]. Duas variações deste tipo de ataque são: a que correlaciona o tempo de execução aos acertos e faltas na *cache*; e a que monitora a sequência de acertos e faltas pelo consumo de energia da CPU [6]. A viabilidade destes ataques muitas vezes requer a execução concorrente de código malicioso para o monitoramento do sistema alvo. Na implementação em Java do AES, a pré-computação e carga antecipada das tabelas seriam uma medida de proteção contra os ataques de *cache*, se não fossem potencialmente desfeitas pelas diversas transformações de código até a sua execução em um hardware específico.

1.6. Considerações finais

Este texto é um primeiro passo na direção da construção de um arcabouço que dê ao programador não especialista a confiança necessária para a codificação segura de métodos criptográficos. Como todos sabemos, o provimento de segurança é uma tarefa sem fim, sob constante ameaça de novas tecnologias e do talento de adversários mais ou menos bem intencionados. Por isso mesmo este texto não pode pretender ser completo e estático. Mais do que os elementos específicos (contramedidas, recomendações, etc), nosso objetivo foi despertar o leitor para a necessidade de implementações cuidadosas de métodos criptográficos, ilustrando com exemplos a diferença entre um código real e aqueles encontrados em livros-textos da área. Esperamos que o leitor programador use este texto como um primeiro recurso para a sua atividade, e que desfrute da sua leitura tanto quanto desfrutamos da sua escrita.

A elaboração deste texto foi desafiadora porque exigiu não apenas os conhecimentos técnicos em criptografia e programação, mas também a habilidade para apresentar os conceitos do modo mais adequado ao público alvo. Um senso de utilidade prática esteve presente durante toda a elaboração do texto. Por isto, muitas vezes, quando foram tomadas decisões de compromisso entre complexidade do conteúdo e facilidade de apresentação, a última sempre foi preferida em detrimento da primeira. A elaboração de exemplos práticos dos maus usos mais comuns em criptografia pode auxiliar programadores em geral a melhorar a segurança de seus softwares. Em um desdobramento interessante, a coleção de maus usos contida neste texto pode dar origem a uma lista de verificação, de apoio à inspeção de código fonte, com grande utilidade em verificações de segurança realizadas sobre softwares criptográficos.

Como era de se esperar em um texto curto (cerca de 50 páginas apenas) para um assunto abrangente e complexo, houve grandes omissões propositais. As ausências mais evidentes, motivadas pelas faltas tanto de espaço no texto quanto de tempo para elaborar uma apresentação adequada, são os exemplos programáticos de acordo de chaves e de validação de certificados digitais. Houve casos em que não foram encontradas, na infraestrutura de software escolhida, implementações fáceis de usar em demonstrações, como foi o caso do acordo de chaves com Diffie-Hellman autenticado.

Além disso, há ainda diversas práticas (boas e ruins) de programação criptográfica que não foram incluídas e poderão, em conjunto às omissões supracitadas, dar origem a um segundo volume deste texto. Citam-se os seguintes assuntos de interesse: a escolha de parâmetros para geração de chaves para algoritmos assimétricos, a utilização programática de curvas elípticas, os bons usos de encriptação determinística em buscas sobre dados cifrados e práticas boas e ruins em certificação digital. Naturalmente, a listagem não pode ser exaustiva.

Finalmente, um último comentário sobre a bibliografia utilizada na elaboração deste texto. Além de respaldar os conceitos apresentados no texto, a bibliografia foi escolhida e organizada para servir como material de leitura para o leitor interessado em aprofundar os estudos. Assim, a lista bibliográfica está dividida nos seguintes tópicos: *(i)* livros texto sobre criptografia moderna, *(ii)* aspectos de implementação criptográfica, *(iii)* programação criptográfica, *(iv)* maus usos da criptografia, *(v)* normas e padrões, *(vi)* vulnerabilidades e ataques e *(vii)* softwares criptográficos. Espera-se, com esta organização, facilitar a navegação do leitor pelo tema e também os estudos futuros.

1.7. Agradecimentos

Alexandre Braga agradece à UNICAMP pelo apoio institucional e financeiro e ao CPqD pelo apoio institucional às atividades acadêmicas de seus funcionários. Ricardo Dahab agradece a UNICAMP e a Universidade de Waterloo pelo apoio institucional, e à Fapesp, Capes e CNPq por auxílios à pesquisa.

1.8. Referências

1.8.1. Criptografia moderna

[1] A. Menezes, P. Van Oorschot, and S. Vanstone, Handbook of applied cryptography. CRC press, 1996.

[2] J. Katz and Y. Lindell, "Introduction to Modern Cryptography," 2006.

[3] W. Mao, *Modern cryptography: theory and practice*. 2003.

[4] W. Stallings, Cryptography and network security, principles and practices. 2003.

1.8.2. Aspectos de implementação criptográfica

[5] C. Koç, About Cryptographic Engineering. 2009.

[6] N. Ferguson, B. Schneier, and T. Kohno, Cryptography Engineering: Design Principles and Practical Applications. Wiley, 2011.

[7] P. Gutmann, "Everything you Never Wanted to Know about PKI but were Forced to Find Out." [Online] https://www.cs.auckland.ac.nz/~pgut001/pubs/pkituto rial.pdf.

[8] P. Gutmann, "Godzilla crypto tutorial - Part 2, Key Management and Certificates." [Online]. Available: https://www.cs.auckland.ac.nz/~pgut001/tutorial/index.html.

1.8.3. Programação criptográfica

[9] A. Braga, C. Rubira, and R. Dahab, "Tropyc: A pattern language for cryptographic object-oriented software, Chapter 16 in Pattern Languages of Program Design 4 (N. Harrison, B. Foote, and," in Also in Procs. of PLoP, 1999.

[10] CryptoWorkshop and BouncyCastle, The Cryptoworkshop Guide to Java Cryptography and the Bouncy Castle APIs. 2013.

[11] D. Hook, Beginning cryptography with Java. John Wiley & Sons, 2005.

[12] T. S. Denis, Cryptography for Developers. Syngress Publishing, 2006.

1.8.4. Maus usos da criptografia

[13] B. Schneier, "Cryptographic design vulnerabilities," Comp., Sept., pp.29–33, 1998.

[14] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why Does Cryptographic Software Fail?: A Case Study and Open Problems," in 5th Asia-Pacific Workshop on Systems, 2014, pp. 7:1–7:7.

[15] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," ACM SIGSAC conference on Computer & communications security - CCS '13, pp. 73–84, 2013.

[16] M. Georgiev, S. Iyengar, and S. Jana, "The most dangerous code in the world: validating SSL certificates in non-browser software," in Proceedings of the 2012 ACM conf. on Computer and comm.. security - CCS '12 (2012), 2012, pp. 38–49.

[17] P. Gutmann, "Lessons Learned in Implementing and Deploying Crypto Software," Usenix Security Symposium, 2002.

[18] R. Anderson, "Why cryptosystems fail," in 1st ACM conference on Computer and communications security, 1993, pp. 215–227.

[19] S. Fahl, M. Harbach, and T. Muders, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," in ACM conference on Computer and communications security, 2012, pp. 50–61.

[20] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications," in IEEE 12th Intl. Conf. Dependable, Autonomic and Secure Computing, 2014, pp. 75–80.

1.8.5. Normas e padrões

[21] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0 (RFC 2898)." [Online]. Available: http://tools.ietf.org/html/rfc2898.

[22] J. Jonsson and Burt Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1," RSA Laboratories, 2003. [Online]. Available: https://tools.ietf.org/html/rfc3447.

[23] NIST, "Advanced Encryption Standard (AES)," NIST FIPS PUB 197, 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[24] NIST, "Cryptographic Algorithm Validation Program (CAVP)." [Online]. Available: csrc.nist.gov/groups/STM/cavp/index.html.

[25] NIST, "Digital Signature Standard (DSS)," NIST FIPS PUB 186-4, 2013. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[26] NIST, "Recommendation for Block Cipher Modes of Operation," NIST SP 800-38A,2001.[Online]. At: csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[27] NIST, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," NIST SP 800-38D, 2007. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[28] NIST, "Recommendation for Key Management – Part 1: General (Revision 3)," NIST SP 800-57, 2012. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf.

[29] NIST, "Secure Hash Standard (SHS)," NIST FIPS PUB 180-4, 2015. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[30] NIST, “The Keyed-Hash Message Authentication Code (HMAC),” NIST FIPS PUB 198-1, 2008. [Online]. Available: http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf.

1.8.6. Vulnerabilidades e ataques

[31] “The Heartbleed Bug.” [Online]. Available: http://heartbleed.com/.

[32] Apple’s SSL/TLS ‘Goto fail’ bug.” [Online]. Available: www.imperialviolet.org/2014/02/22/applebug.html.

[33] J. Rizzo and T. Duong, “Practical padding oracle attacks,” Proceedings of the 4th USENIX conference on Offensive technologies (2010), pp. 1–9, 2010.

[34] Piercing Through WhatsApp’s Encryption.” [Online]. Available: https://blog.thijsalkema.de/blog/2013/10/08/piercing-through-whatsapp-s-encryption/.

[35] S. Vaudenay, “Security Flaws Induced by CBC Padding—Applications to SSL, IPSEC, WTLS...,” Advances in Cryptology—EUROCRYPT 2002, no. 1, 2002.

[36] T. Duong and J. Rizzo, “Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET,” IEEE Security and Privacy, pp. 481–489, 2011.

1.8.7. Softwares criptográficos

[37] “EJBCA - Open source PKI and CA.” [Online]. Available: www.ejbca.org.

[38] “Java Cryptography Architecture (JCA) Reference Guide.” [Online]. Available: docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html.

[39] “The Legion of the Bouncy Castle.” [Online]. Avail.: http://www.bouncycastle.org.

[40] “OpenSSL Cryptography and SSL/TLS toolkit.” [Online]. Avail.: OpenSSL.org.